

Swift

游戏开发案例实战

刘媛媛 编著

国内第一本专门介绍Swift游戏开发的图书

155个示例、5个游戏项目案例，详解Swift 游戏开发的各项关键技术和流程

- ☑ 环境搭建 → 开发第一个App → 掌握Swift关键语法 → 游戏项目案例实战，逐步掌握Swift 游戏开发的各项关键技术
- ☑ 详解5个游戏项目案例：记忆配对、太空侵略者、Simon记忆、迷你高尔夫和银河大战
- ☑ 结合游戏案例，详解场景切换、绘制图像、游戏引擎、音频引擎和用户交互等关键技术
- ☑ 展示游戏开发的全流程：环境搭建 → 绑定账号 → 应用开发 → 应用测试 → 应用发布
- ☑ 提供了QQ群、技术论坛和E-mail等完善的学习交流和沟通方式



(附赠51CTO学院学习卡)



清华大学出版社

Swift

游戏开发案例实战

本书涵盖的精华内容

- ✓ 搭建开发环境——Xcode的安装与运行
- ✓ 编写第一个Swift程序
- ✓ Swift基础语法
- ✓ Swift高级语法
- ✓ iPhone游戏开发基础——记忆配对游戏
- ✓ 太空侵略者——绘制图像
- ✓ 太空侵略者2——游戏引擎
- ✓ Simon记忆游戏——音频引擎
- ✓ 迷你高尔夫——用户交互
- ✓ 银河大战——Sprite Kit游戏引擎和传感器应用
- ✓ 应用程序的发布

本书配套资源获取方式

本书涉及的源程序等资源需要读者自行下载。请到清华大学出版社的网站上搜索到本书页面，然后按照提示下载即可。读者也可以在本书服务网站上的相关版块下载。具体见本书前言中的说明。



清华大学出版社数字出版网站

WQBook  书文局泉
www.wqbook.com

上架建议：计算机/移动开发

ISBN 978-7-302-39575-1



9 787302 395751 >

定价：59.80元

Swift

游戏开发案例实战

刘媛媛 编著



清华大学出版社

内 容 简 介

本书是国内第一本 Swift 游戏开发图书。本书由浅入深、全面、系统地讲解了 Swift 游戏开发的基础知识和各项关键技术,其中重点介绍了 5 个游戏项目案例的开发,供读者实战演练。同时,本书也提供了这些游戏案例的完整源代码,供读者学习和研究。

本书共 11 章。其中,第 1~4 章主要介绍了开发环境的搭建、账号绑定、模拟器的使用、真机测试和 Swift 编程必备基础知识等。第 5~11 章以游戏项目案例为主导,讲解了记忆配对、太空侵略者、Simon 记忆游戏、迷你高尔夫和银河大战 5 个游戏项目案例的开发过程和应用程序的发布。在讲解过程中,对游戏开发中的核心关键技术进行了仔细的讲解。这些技术包括图像绘制、游戏引擎、音频引擎、用户交互、Sprite Kit、传感器应用和 App 的发布等。

本书涉及面广,从基本的操作到游戏开发的关键技术,再到游戏项目案例实战,几乎涉及 Swift iOS 游戏开发的各方面的重要知识。本书不仅适合游戏开发爱好者和游戏开发一线程序员阅读,也适合 Swift 初学者和 iOS 各类开发人员阅读。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

Swift 游戏开发案例实战 / 刘媛媛编著. —北京:清华大学出版社, 2015

ISBN 978-7-302-39575-1

I. ①S… II. ①刘… III. ①游戏程序—程序设计 IV. ①TP311.5

中国版本图书馆 CIP 数据核字(2015)第 046535 号

责任编辑:杨如林

封面设计:欧振旭

责任校对:徐俊伟

责任印制:何 芊

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座 邮 编:100084

社总机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者:清华大学印刷厂

装 订 者:三河市吉祥印务有限公司

经 销:全国新华书店

开 本:185mm×260mm 印 张:22.75 字 数:565 千字

版 次:2015 年 5 月第 1 版 印 次:2015 年 5 月第 1 次印刷

印 数:1~3500

定 价:59.80 元

产品编号:063388-01

前言

2014 年 9 月，苹果公司推出了新一代开发语言 Swift。该语言开发效率更高，受到了苹果开发者的广泛欢迎。在国内，大量的开发人员也开始逐渐从 Objective-C 转向了 Swift 语言，所以使用 Swift 语言进行开发的应用也越来越多。而游戏类 App 由于其用户数量巨大，更容易为开发者带来可观的收益，所以游戏开发，尤其是基于手机的游戏开发越来越受到广大开发人员的青睐。

在所有的 App 开发类别中，游戏开发不同于普通应用开发。它在软件的图形、音频和交互等方面都有特殊的要求。这些技术就成为了开发者所必须要掌握的知识。本书从 Swift 语言的基础开始讲解，然后重点以案例的形式讲解了如何使用 Swift 语言开发苹果游戏应用。书中着重讲解了 5 项核心技术，帮助读者快速具备游戏应用开发的技能。

本书特色

1. 内容由浅入深

Swift 作为一门新兴语言，大部分开发者往往对其没有足够的了解。本书通过两章的内容帮助读者巩固 Swift 的语言基础，更快地融入 Swift 开发工作中。

2. 以游戏项目案例为主导

本书详细讲解了 5 个游戏的开发过程，如记忆配对、太空侵略者、Simon 记忆、迷你高尔夫和银河大战。这些项目将帮助读者更好地理解 iOS 项目开发的方式和流程。

3. 涵盖游戏开发的关键技术

游戏开发不同于普通软件的开发，它对图形、音频和交互等方面有更高的要求。本书结合案例，着重讲解这些重要的技术，并且最后还讲解 SpriteKit 的使用。

4. 贴近实际开发

本书全面介绍了实际苹果应用开发的各个环节，内容包括环境搭建、绑定苹果账号、游戏应用开发和应用程序发布。通过完整的开发过程呈现，可以帮助读者更好的掌握开发过程。

本书内容及体系结构

第 1 章 开发环境搭配——Xcode 的安装与运行

本章主要内容包括苹果账号的注册、Xcode 的下载和安装、首次打开 Xcode，以及 Xcode

的界面介绍等内容。通过本章的学习，读者可以了解和掌握如何去注册一个苹果账号，以及如何去创建项目。

第 2 章 编写第一个 Swift 程序

本章主要内容包括程序的运行、模拟器的操作、编辑界面的介绍、代码编写以及真机测试等内容。通过本章的学习，读者可以详细的了解和掌握在创建的项目中编写简单的 Swift 程序，以及实现真机测试。

第 3 章 Swift 基础语法

本章主要内容包括常量、变量、数据类型、字面值、运算符、类型转换、字符串、集合类型、程序控制结构、函数和闭包等内容。通过本章的学习，读者可以详细的了解和掌握 Swift 处理数据的各种方法。

第 4 章 Swift 高级语法

本章主要内容包括类、继承、枚举、结构、构造方法和析构方法、扩展和协议、运算符重载和泛型等内容。通过本章的学习，读者可以详细的了解和掌握 Swift 的一些高级语法的内容。

第 5 章 iPhone 游戏开发基础——记忆配对游戏

本章主要内容包括游戏介绍、创建项目、主菜单模块、配对模板、分数榜单模块的设计和场景切换等内容。通过本章的学习，读者可以详细的了解和掌握如何进行界面设计、在翻转两个卡牌后的游戏配对，以及场景切换等内容。

第 6 章 太空侵略者——绘制图像

本章主要内容包括游戏中飞船的创建和移动、敌人的创建和移动、子弹的创建和移动等内容。通过本章的学习，读者可以详细的了解和掌握视图的创建以及移动。

第 7 章 太空侵略者 2——游戏引擎

本章主要内容包括提示界面的设计，以及检测碰撞等内容。通过本章的学习，读者可以详细的了解和掌握如何实现两个物体的碰撞功能。

第 8 章 Simon 记忆游戏——音频引擎

本章主要内容包括游戏的界面设计、添加颜色提示序列和添加背景音乐等内容。通过本章的学习，读者可以详细的了解和掌握如何实现声音的添加和使用。

第 9 章 迷你高尔夫——用户交互

本章主要内容包括游戏的界面设计、用户交互中的不足和杆数显示等内容。通过本章的学习，读者可以详细的了解和掌握游戏中用户交互的实施方式。

第 10 章 银河大战——Sprite Kit 游戏引擎和传感器应用

本章主要内容包括创建 Game 类型项目、游戏的界面设计、什么是 Sprite Kit，以及使

用传感器操控飞船等内容。通过本章的学习，读者可以详细的了解和掌握如何使用 Sprite Kit 去开发游戏，以及如何使用传感器。

第 11 章 应用程序的发布

本章主要内容包括创建 App ID、申请发布证书、准备提交应用程序、提交应用程序到 App Store 上，以及常见审核不通过的原因等内容。通过本章的学习，读者可以详细的了解和掌握如何将一个应用程序上传到 App Store 上。

本书读者对象

- ☐ Swift 编程初学者；
- ☐ iOS 游戏开发人员；
- ☐ iOS 开发爱好者；
- ☐ 大中专院校的学生；
- ☐ 社会培训班的学员。

学习建议

- ☐ 坚持编程。编程需要大量的练习。就像学习英语一样，只有不停地练习才能掌握英语的使用，所以只有不停地练习编写程序才能掌握好编程。
- ☐ 随时实践。学习时，可能脑子里随时会冒出很多想法，大胆使用程序想办法去实现这些想法，从中获取成就感，这会成为你持续学习的动力。
- ☐ 相互交流和沟通。一个人学到的和想到的东西总是有限的，只有相互交流和沟通才能对一个知识点有更加全面和深入的理解。

本书约定

本书使用的 Xcode 版本是 6.0.1 版本。如果读者使用 Xcode 的其他版本，可能会发生编译错误的问题。所以，建议读者最好基于 Xcode 6.0.1 的开发环境来学习本书内容。

本书配套资源获取方式

本书提供以下的配套资源：

- ☐ 本书实例源代码；
- ☐ 本书开发环境。

为了节省读者的购书开支，本书放弃以配书光盘的方式提供这些资源，而是改为采用提供下载的方式。读者可以在本书的服务网站（www.wanjuanchina.net）的相关版块上下载这些配套资源。另外，清华大学出版社的网站上（www.tup.com.cn）也提供了本书的源程序，以方便读者下载。读者可以在该网站上搜索到本书页面，然后按照提示下载。

本书售后服务方式

编程学习的最佳方式是共同学习。但是由于环境所限，大部分读者都是独自前行。为了便于读者更好地学习 Swift 语言，我们构建了多样的学习环境，力图打造立体化的学习方式，除了对内容精雕细琢之外，还提供了完善的学习交流和沟通方式。主要有以下几种方式：

- ❑ 提供技术论坛 <http://www.wanjuanchina.net>，读者可以将学习过程中遇到的问题发布到论坛上以获得帮助。
- ❑ 提供 QQ 交流群 336212690，读者申请加入该群后便可以和作者及广大读者交流学习心得，解决学习中遇到的各种问题。
- ❑ 提供 mmbbc@wanjuanchina.net 和 bookservice2008@163.com 服务邮箱，读者可以将自己的疑问发电子邮件以获取帮助。

本书作者

本书主要由刘媛媛编写。其他参与编写的人员有陈冠军、陈浩、黄振东、蒋庆学、李代叙、李世民、李思清、李云龙、李志刚、刘存勇、刘燕珍、龙哲、吕轶、牟春梅、屈明环、石峰、史艳艳、宋宁宁、王德亮、王俊清、王雅宁、翁盛鑫。

阅读本书的过程中若有任何疑问，都可以发邮件或者在论坛和 QQ 群里提问，会有专人为您解答。最后顺祝各位读者读书快乐！

编者

目 录

第 1 章	开发环境搭配——Xcode 的安装与运行	1
1.1	苹果账号	1
1.1.1	苹果账号的成员分类	1
1.1.2	注册免费的苹果账号	1
1.1.3	注册非免费的苹果账号	4
1.2	Xcode 的下载和安装	6
1.2.1	App Store 中下载和安装 Xcode	6
1.2.2	其他网站下载和安装 Xcode	9
1.3	绑定苹果账号	10
1.4	更新组件和文档	12
1.5	首次打开 Xcode	13
1.6	Xcode 的界面介绍	15
1.6.1	导航窗口	15
1.6.2	工具窗口	15
1.6.3	编辑窗口	17
1.6.4	目标窗口	17
第 2 章	编写第一个 Swift 程序	19
2.1	运行程序	19
2.2	模拟器的操作	21
2.2.1	模拟器与真机的区别	21
2.2.2	退出应用程序	21
2.2.3	应用程序图标的设计	21
2.2.4	语言设置	23
2.2.5	旋转	26
2.2.6	删除应用程序	27
2.3	编辑界面	27
2.3.1	界面介绍	27
2.3.2	设计界面	29
2.3.3	视图对象库的介绍	31
2.4	编写代码	33
2.5	调试	35
2.6	真机测试	38

2.6.1	申请和下载证书	38
2.6.2	实现真机测试	47
2.7	使用帮助文档	47
第 3 章	Swift 基础语法	49
3.1	常量和变量	49
3.1.1	常量	49
3.1.2	变量	51
3.2	数据类型	52
3.2.1	整数类型	52
3.2.2	浮点类型	54
3.2.3	字符类型	55
3.2.4	布尔类型	55
3.2.5	可选类型	56
3.2.6	类型别名	56
3.3	值的表示——字面值	57
3.3.1	整型字面值	57
3.3.2	浮点型字面值	57
3.3.3	字符型字面值	58
3.3.4	字符串型字面值	58
3.3.5	布尔型字面值	59
3.3.6	元组型字面值	59
3.4	运算符	60
3.4.1	元的介绍	60
3.4.2	赋值运算符	60
3.4.3	一元加运算符	61
3.4.4	一元减运算符	61
3.4.5	算术运算符	61
3.4.6	自增、自减运算符	62
3.4.7	比较运算符	63
3.4.8	逻辑运算符	64
3.4.9	位运算符	65
3.4.10	复合运算符	66
3.4.11	求字节运算符	67
3.4.12	强制解析运算符	67
3.4.13	区间运算符	68
3.4.14	溢出运算符	69
3.5	类型转换	70
3.5.1	整数的转换	70
3.5.2	整数与浮点数的转换	70

3.6	字符串	71
3.6.1	字符串的初始化	71
3.6.2	字符串的操作	71
3.7	集合类型	72
3.7.1	数组	72
3.7.2	字典	73
3.8	程序控制结构	75
3.8.1	顺序结构	75
3.8.2	选择结构	75
3.8.3	循环结构	80
3.8.4	跳转语句	83
3.8.5	标签语句	85
3.9	函数	86
3.9.1	函数的介绍	87
3.9.2	无参函数的使用	87
3.9.3	有参函数的使用	88
3.9.4	函数的参数的注意事项	89
3.9.5	函数的返回值	92
3.9.6	函数类型	94
3.9.7	常用的标准函数	96
3.9.8	函数的嵌套	97
3.10	闭包	99
3.10.1	闭包表达式	99
3.10.2	Trailing 闭包	102
3.10.3	捕获值	103
第 4 章	Swift 高级语法	105
4.1	类	105
4.1.1	创建类	105
4.1.2	实例化对象	105
4.1.3	属性	106
4.1.4	方法	110
4.1.5	下标脚本	112
4.1.6	类的嵌套	114
4.1.7	可选链接	117
4.2	继承	118
4.2.1	继承的实现	118
4.2.2	重写	120
4.2.3	禁止重写	124
4.2.4	类型检测	124

4.3	枚举	127
4.3.1	定义枚举	128
4.3.2	定义枚举成员	128
4.3.3	实例化枚举的对象	129
4.3.4	枚举成员与 switch 语句的匹配	130
4.3.5	访问枚举中成员的原始值	131
4.3.6	相关值	132
4.3.7	定义枚举类型的其他	132
4.3.8	枚举的嵌套	134
4.4	结构	136
4.4.1	定义结构	136
4.4.2	实例化结构对象	137
4.4.3	在结构中定义内容	137
4.5	构造方法和析构方法	140
4.5.1	值类型的构造器	140
4.5.2	类的构造器	143
4.5.3	析构方法	147
4.6	扩展和协议	148
4.6.1	扩展	148
4.6.2	协议	151
4.6.3	可选协议	155
4.6.4	使用协议类型	156
4.6.5	协议的继承	157
4.6.6	协议的组合	158
4.6.7	检查协议的一致性	159
4.6.8	委托	160
4.7	运算符重载	161
4.7.1	算术运算符重载	162
4.7.2	前置运算符和后置运算符重载	162
4.7.3	复合运算符重载	163
4.7.4	比较运算符重载	164
4.7.5	自定义运算符	165
4.8	泛型	168
4.8.1	泛型函数	168
4.8.2	泛型类型	169
4.8.3	具有多个类型参数的泛型	171
4.8.4	类型约束	171
4.8.5	关联类型	172
第 5 章	iPhone 游戏开发基础——记忆配对游戏	176
5.1	游戏介绍	176

5.2	开发游戏之前的准备工作	177
5.2.1	创建项目	177
5.2.2	添加图像	178
5.3	主菜单模块	179
5.4	配对模块	182
5.4.1	设计界面	183
5.4.2	卡牌的翻转	186
5.5	核心功能——卡牌的配对	194
5.5.1	翻转两个卡牌	194
5.5.2	判断卡牌	194
5.5.3	配对成功和失败的操作	195
5.5.4	完成所有配对	197
5.6	配对模块的额外功能	199
5.6.1	猜测次数功能	199
5.6.2	提高游戏的难度	200
5.7	分数榜单模块	202
5.7.1	准备工作	202
5.7.2	界面设计	202
5.7.3	实现分数的显示	205
5.8	关于游戏模块	209
5.9	场景切换	210
5.9.1	什么是场景切换	210
5.9.2	实现场景切换	211
5.9.3	过渡动画效果	213
5.9.4	全部的场景切换	214
第 6 章	太空侵略者——绘制图像	217
6.1	游戏介绍	217
6.2	开发游戏之前的准备工作	218
6.3	主菜单模板	218
6.4	射击游戏模板	219
6.4.1	准备工作	219
6.4.2	设计界面	220
6.5	添加飞船	221
6.6	移动飞船	223
6.6.1	向左移动	223
6.6.2	向右移动	224
6.7	创建敌人	225
6.7.1	创建单个敌人的创建	225
6.7.2	创建多个敌人	226

6.8	移动敌人	228
6.9	发射子弹	229
6.9.1	飞船的子弹	229
6.9.2	敌人的子弹	231
6.10	场景的切换	234
第 7 章	太空侵略者 2——游戏引擎	236
7.1	游戏介绍	236
7.2	开发游戏前的准备工作	237
7.3	主菜单模块	238
7.4	射击游戏模块	239
7.5	了解状态机	240
7.6	使用代码添加射击游戏界面元素	241
7.6.1	提示界面	241
7.6.2	飞船	243
7.6.3	敌人	248
7.7	检测碰撞	253
7.7.1	敌人的子弹击中飞船的检测	253
7.7.2	飞船的子弹击中敌人的检测	255
7.8	计分功能	256
7.9	歼灭所有敌人	257
7.10	分数榜模块	259
7.10.1	准备工作	259
7.10.2	界面设计	259
7.10.3	实现分数的显示	260
7.11	场景切换	265
第 8 章	Simon 记忆游戏——音频引擎	268
8.1	游戏介绍	268
8.2	开发游戏之前的准备工作	269
8.3	主菜单模块	272
8.4	游戏模块	273
8.4.1	准备工作	273
8.4.2	界面设计	274
8.5	添加颜色提示序列	275
8.5.1	添加提示声音	275
8.5.2	添加颜色提示	278
8.6	玩家猜测	279
8.7	添加背景音乐	281
8.8	游戏模块的额外功能	282
8.8.1	显示游戏处于的关数	282

8.8.2	提高游戏的难度	283
8.9	游戏介绍模块	285
8.10	场景切换	286
第 9 章	迷你高尔夫——用户交互	288
9.1	游戏介绍	288
9.2	开发游戏之前的准备工作	289
9.3	主菜单模块	290
9.4	游戏模块	291
9.4.1	准备工作	291
9.4.2	界面设计	291
9.4.3	添加高尔夫球	292
9.4.4	移动高尔夫球	293
9.5	用户交互中的不足	296
9.5.1	边界的限定	296
9.5.2	速度限定	297
9.5.3	进洞的限定	297
9.6	杆数显示	300
9.7	游戏界面模块	301
9.8	场景切换	302
第 10 章	银河大战——Sprite Kit 游戏引擎和传感器应用	306
10.1	游戏介绍	306
10.2	创建 Game 类型项目	307
10.2.1	Game 模板类型简介	307
10.2.2	创建项目	308
10.2.3	添加图像和音频文件	309
10.3	主菜单模块	310
10.4	射击游戏模块	311
10.5	为射击游戏界面添加元素	312
10.5.1	准备工作	312
10.5.2	什么是 Sprite Kit	313
10.5.3	使用 SKSpriteNode 添加背景	313
10.5.4	使用 SKSpriteNode 添加飞船	315
10.5.5	使用 SKSpriteNode 添加敌人	316
10.6	发射子弹	316
10.6.1	添加子弹	317
10.6.2	通过触摸发射子弹	317
10.7	使用传感器操控飞船	318
10.7.1	传感器介绍	318
10.7.2	判断传感器是否可用	318

10.7.3 实现移动	319
10.8 碰撞检测	322
10.9 分数显示	323
10.9.1 使用 SKLabelNode 添加显示分数的节点	323
10.9.2 实现分数的显示	324
10.10 添加声音	325
10.10.1 进入射击游戏界面的声音	325
10.10.2 子弹击中敌人的声音	326
10.11 游戏介绍模块	326
10.12 场景切换	327
第 11 章 应用程序的发布	330
11.1 创建 App ID	330
11.2 申请发布证书	331
11.2.1 申请证书	331
11.2.2 申请证书对应的配置文件 (Provision File)	333
11.3 准备提交应用程序	335
11.3.1 创建应用及基本信息	335
11.3.2 项目的相关设置	337
11.4 提交应用程序到 App Store 上	343
11.5 常见审核不通过的原因	348

第 1 章 开发环境搭配——Xcode 的安装与运行

每一种应用程序的开发都有自己特定的开发环境。所谓开发环境就是为了支持系统软件和应用软件工程化开发和维护的一组软件。它通常简称为 SDE。iPhone 游戏应用程序的开发需要使用到 Xcode。本章将讲解 Xcode 的安装、苹果账号的注册，以及绑定苹果账号等内容。

1.1 苹果账号

苹果公司专门有一个提供软件下载的在线商店，用户如果想要下载软件，就需要注册一个苹果账号。通常，苹果账号必须是一个邮箱账号，并在苹果网站上注册、验证通过。

1.1.1 苹果账号的成员分类

在苹果公司注册苹果账号，就可以成为开发成员。开发成员一共可以分为四种，如表 1-1 所示。

表 1-1 iPhone苹果账号的成员

成 员 类 型	成 本
在线开发成员	免费
标准 iPhone 开发成员	\$99/年
企业 iPhone 开发成员	\$299/年
大学 iPhone 开发成员	免费

1.1.2 注册免费的苹果账号

以下是注册一个免费的苹果账号的具体步骤。

(1) 在 Dock（Dock 一般指的是苹果操作系统中的停靠栏）中，找到 Safari，如图 1.1 所示。

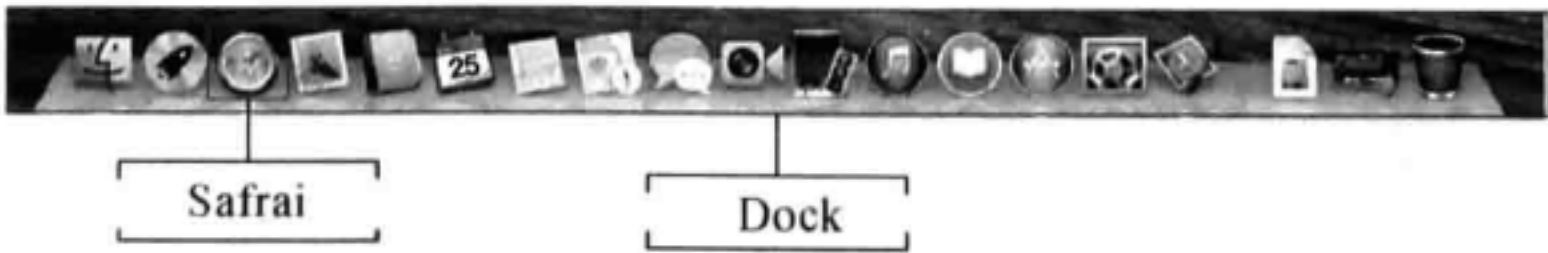


图 1.1 操作步骤 1

(2) 单击打开 Safrai，如图 1.2 所示。



图 1.2 操作步骤 2

(3) 在搜索栏中输入网址（https://developer.apple.com/devcenter/ios/index.action），然后按回车键，进入 iOS Dev Center-App Developer 网页，如图 1.3 所示。



图 1.3 操作步骤 3

(4) 选择 register for free 选项，进入 Apple Developer Registration-Apple Developer 网页，如图 1.4 所示。

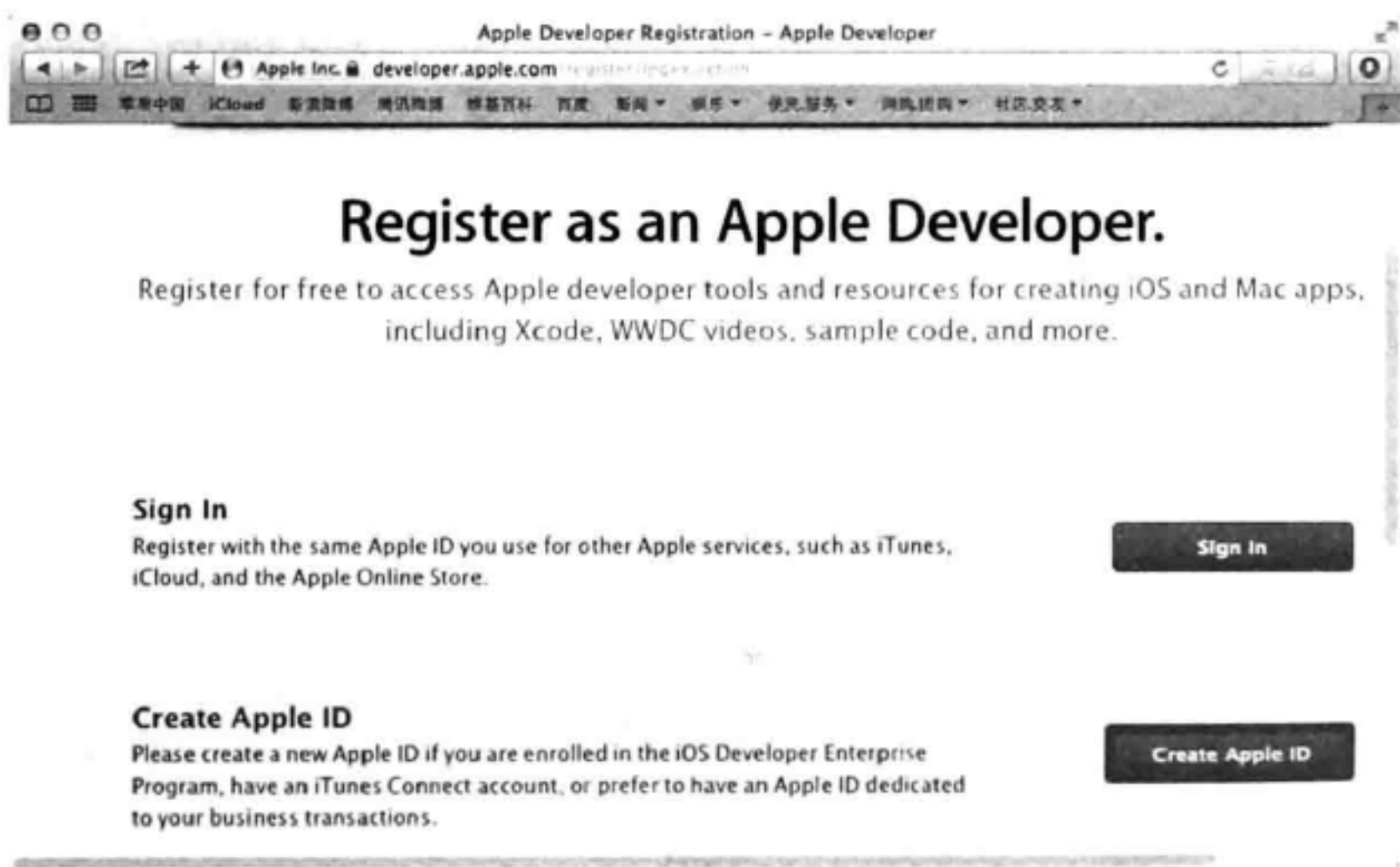


图 1.4 操作步骤 4

(5) 单击 Create Apple ID 按钮，进入 Apple-My Apple ID 网页，如图 1.5 所示。



图 1.5 操作步骤 5

(6) 在网页中按照要求输入内容后，单击网页最下方的 Create Apple ID 按钮，进入确认邮件地址的网页，如图 1.6 所示。



图 1.6 操作步骤 6

(7) 单击 Continue 按钮，进入确定邮件地址的另一个网页。单击此网页中的 Send Verification Email 按钮，苹果公司会向作为账号的邮箱发送一封确认邮件。

(8) 进入账号所使用的邮箱，就会看到 Apple 发来的一封确定邮件地址的邮件。打开该邮件，如图 1.7 所示。

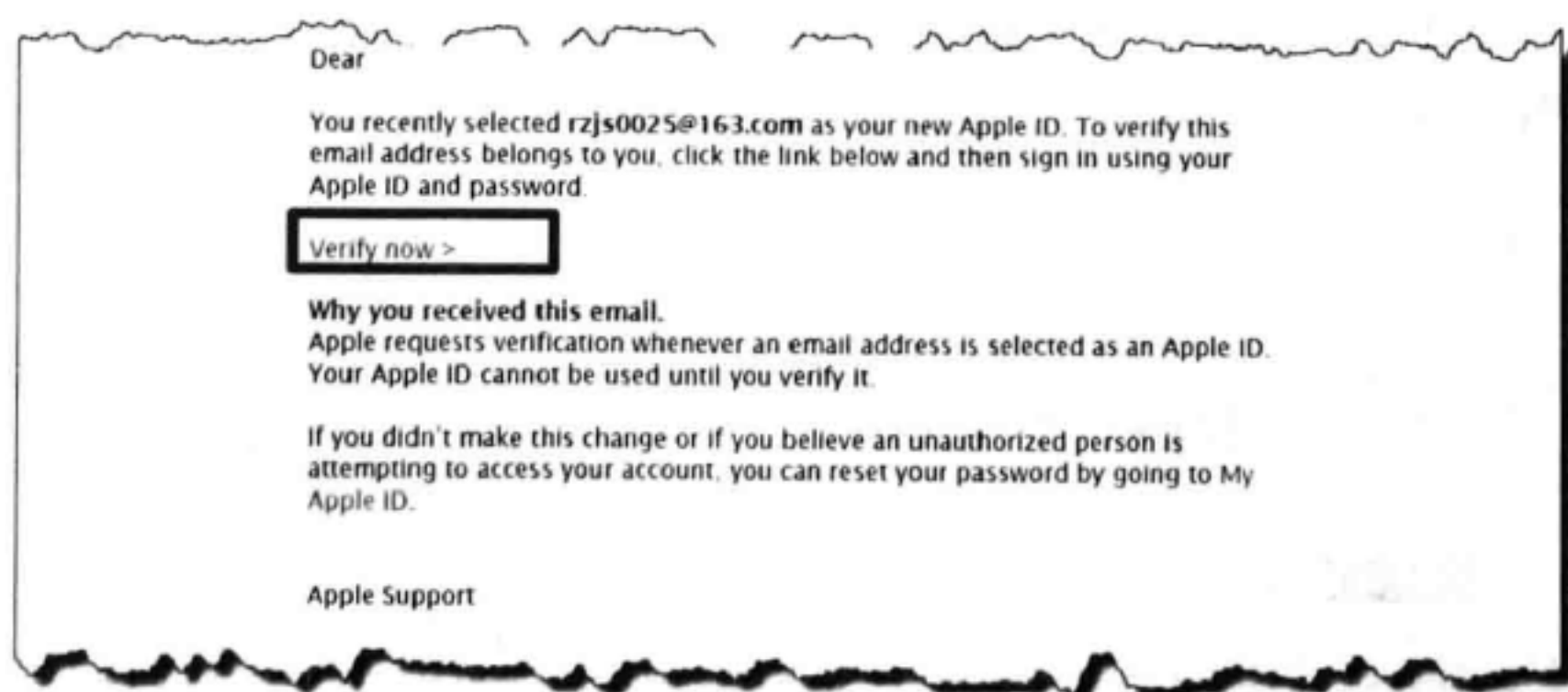


图 1.7 操作步骤 7

(9) 单击 Verify now 链接，进入 Apple-My Apple ID-Email Verification 网页，如图 1.8 所示。



图 1.8 操作步骤 8

(10) 输入需要验证的邮箱以及地址，然后单击 Verify Address 按钮，进入下一个网页，此网页会提示开发者注册的 Apple ID 现在已经可以使用了。

1.1.3 注册非免费的苹果账号

由于免费的苹果账号在开发应用程序时受到了很多的限制，如不可以进行真机测试等。所以需要注册一个非免费的苹果账号。以下是注册非免费的苹果账号的具体步骤。

(1) 在 Safari 中输入网址 (https://developer.apple.com/programs/)，然后按回车键，如图 1.9 所示。



图 1.9 操作步骤 1

(2) 选择 iOS Developer Program 选项，进入 iOS Developer Program-Apple Developer 网页，如图 1.10 所示。

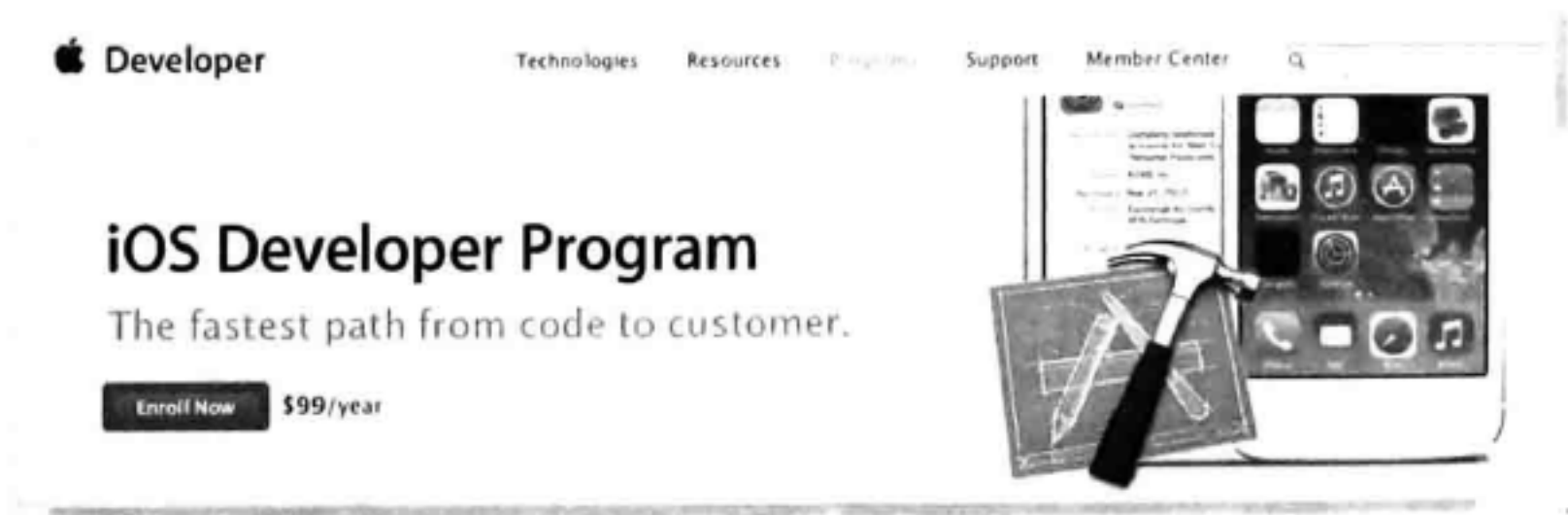


图 1.10 操作步骤 2

(3) 单击 Enroll Now 按钮，进入 Enrolling in Apple Developer Programs-Apple Developer 网页，如图 1.11 所示。

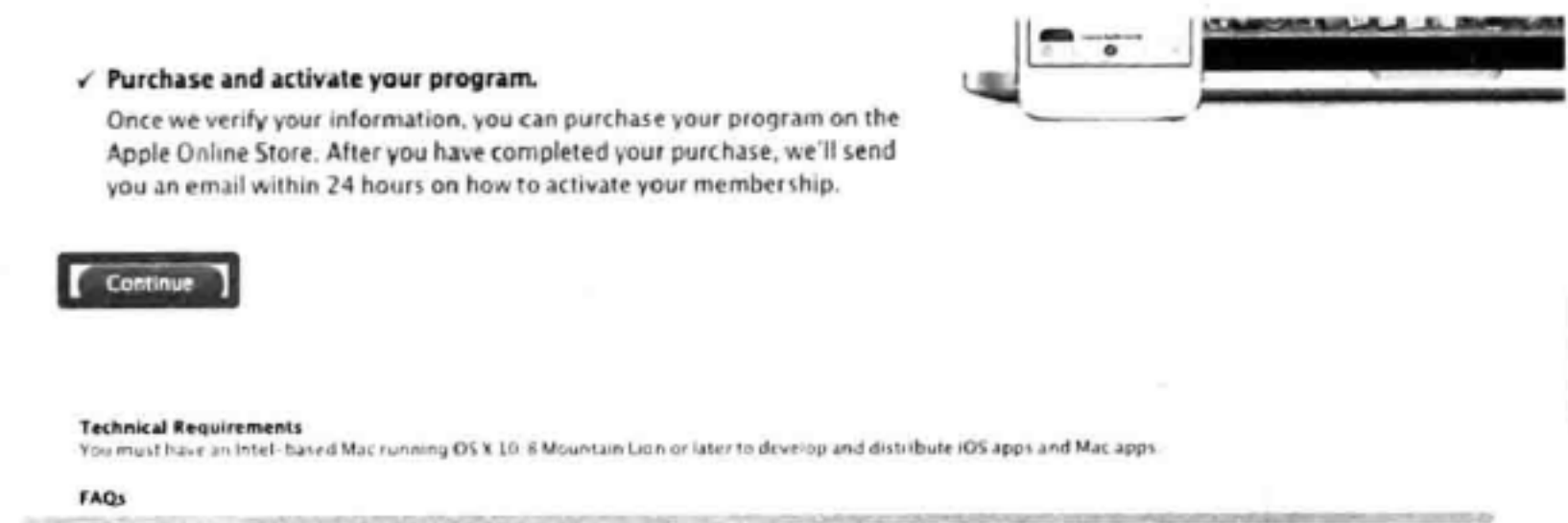


图 1.11 操作步骤 3

(4) 单击 Continue 按钮，进入 Sign in or create an Apple ID-Apple Developer Program Enrollment 网页，如图 1.12 所示。



图 1.12 操作步骤 4

(5) 单击 Sign In 按钮，进入 Apple Developer Program Enrollment 网页，如图 1.13 所示。

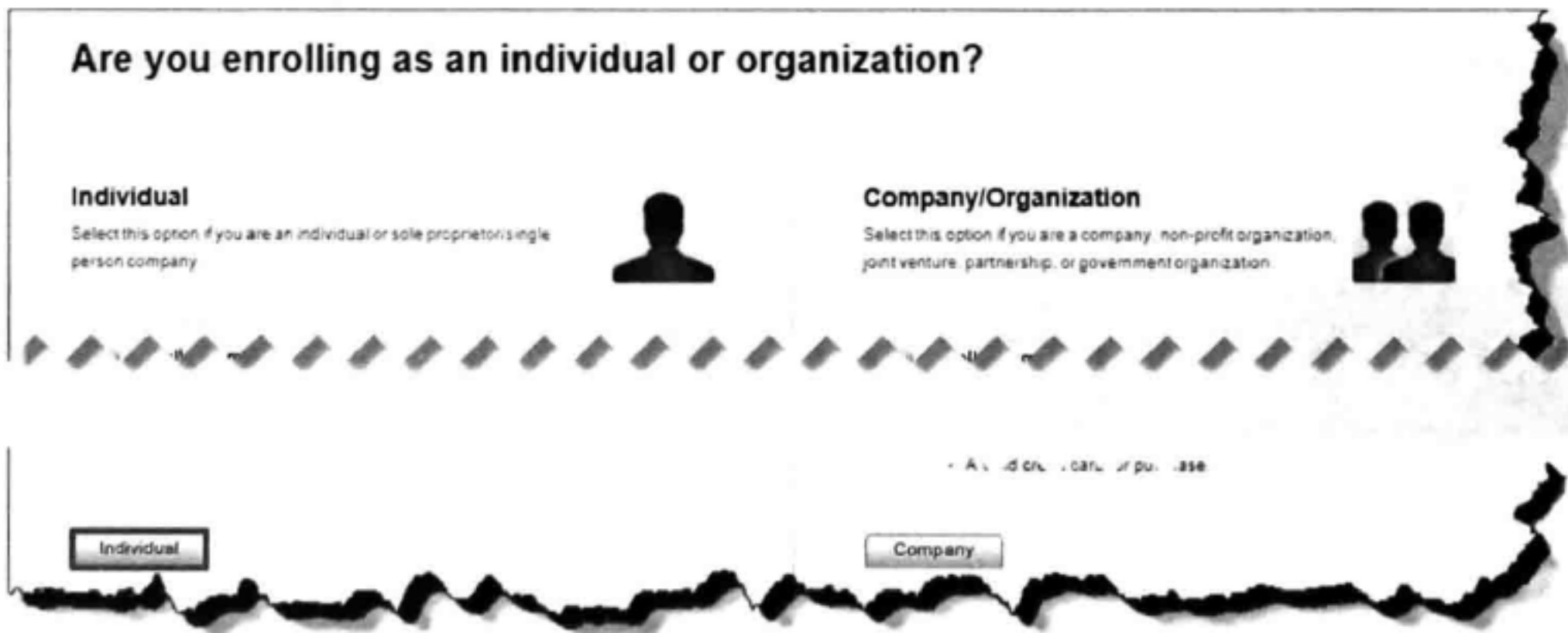


图 1.13 操作步骤 5

(6) 单击 Individual 按钮后，进入 Sign in with your Apple ID-Apple Developer 网页，如图 1.14 所示。

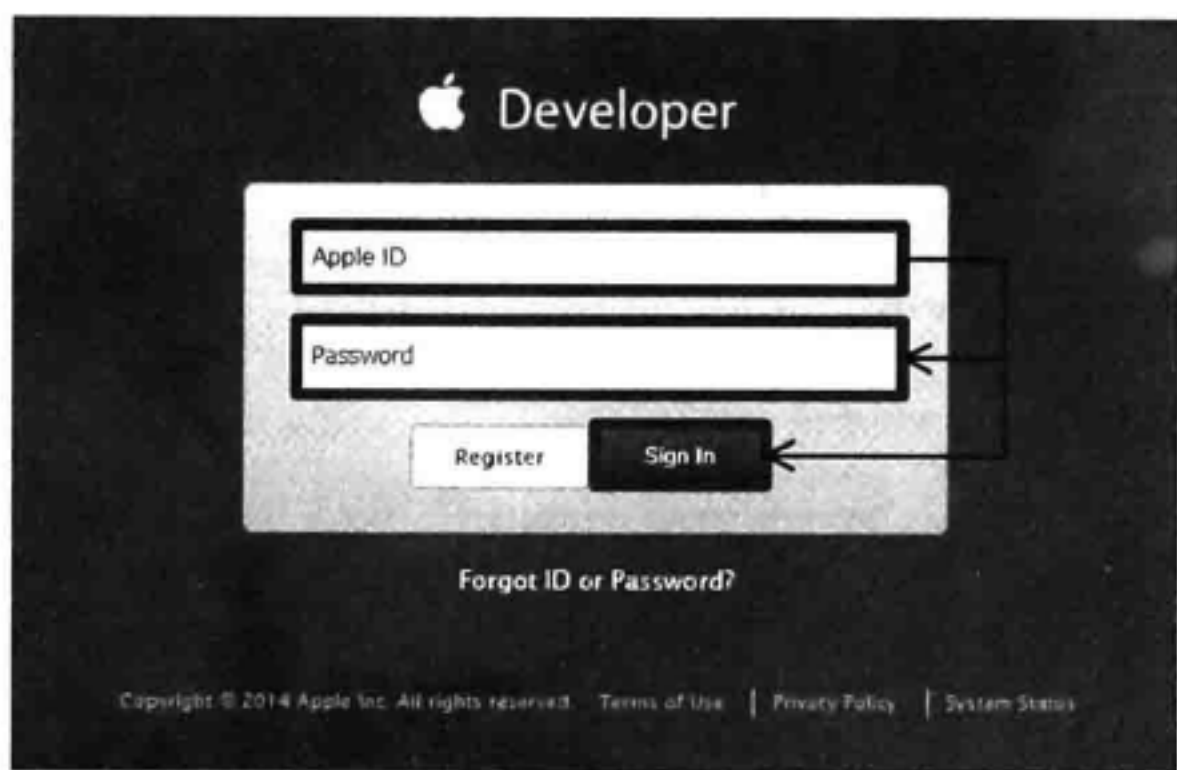


图 1.14 操作步骤 6

(7) 输入 Apple ID 以及密码后，单击 Sign In 按钮，进入 Apple Developer Program Enrollment-Update Information 网页，完善自己的信息，然后单击 Continue 按钮。以上这几步是申请付费开发者账号的重要步骤，剩下的步骤就需要根据开发者的需求进行填写了。这里就不再做介绍了。

注意：从申请一个付费的开发者账号开始到激活大概需要 3~5 天，这段时间需要开发者留心你的与苹果账号关联的邮箱，苹果公司会为此邮箱发一些邮件。

1.2 Xcode 的下载和安装

Xcode 是苹果公司为开发应用程所提供的开发环境。本节将讲解此开发环境的两种下载和安装方式，即在 App Store 中下载和安装 Xcode 和在其他的网站下载和安装 Xcode。

1.2.1 App Store 中下载和安装 Xcode

App Store 中提供了很多的软件，而 Xcode 也在其中。以下就是在 App Store 中下载和安装 Xcode 的具体步骤。

(1) 在 Dock 中找到 App Store，如图 1.15 所示。

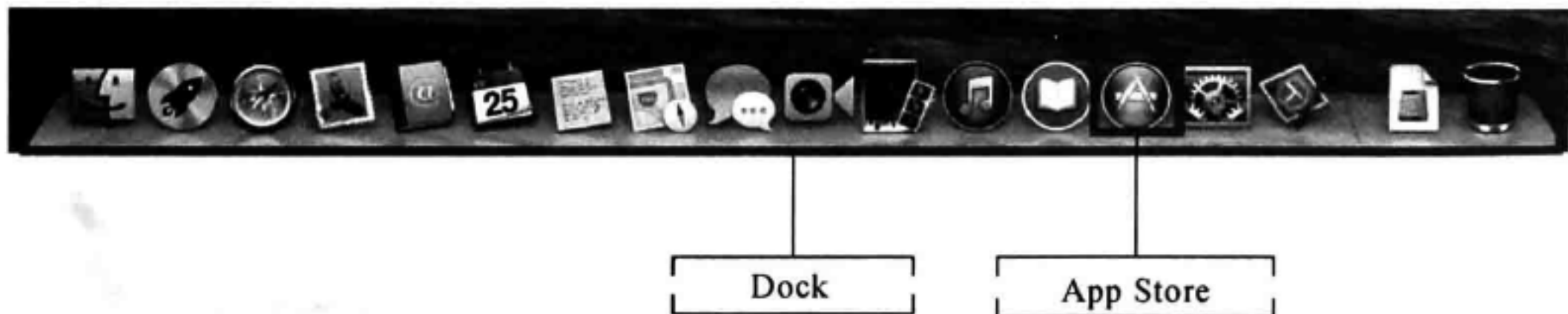


图 1.15 操作步骤 1

(2) 单击 App Store 图标，打开 App Store，如图 1.16 所示。

(3) 在搜索栏中输入要搜索的内容，即 Xcode，然后按回车键，进行搜索，如图 1.17 所示。



图 1.16 操作步骤 2



图 1.17 操作步骤 3

(4) 单击 Xcode 右下方的“免费”按钮，此时“免费”按钮就变为了“安装 APP”按钮，如图 1.18 所示。



图 1.18 操作步骤 4

(5) 单击“安装 APP”按钮，弹出“登录 App Store 来下载”对话框，如图 1.19 所示。
(6) 输入 Apple ID 以及密码后，单击“登录”按钮。此时“安装 APP”按钮变为了“安装”按钮，如图 1.20 所示。并且 Xcode 会在 Launchpad 中进行下载和安装，如图 1.21 所示。



图 1.19 操作步骤 5



图 1.20 操作步骤 6



图 1.21 操作步骤 7

(7) 一般在 Launchpad 中下载的软件，都可以在应用程序中找到。选择“前往”|“应用程序”命令打开应用程序，如图 1.22 所示。



图 1.22 操作步骤 8

- (8) 双击 Xcode，弹出 Xcode and iOS SDK License Agreement 对话框，如图 1.23 所示。
- (9) 单击 Agree 按钮，弹出““Xcode”想要进行更改。键入您的密码以允许执行此操作”对话框，如图 1.24 所示。



图 1.23 操作步骤 9

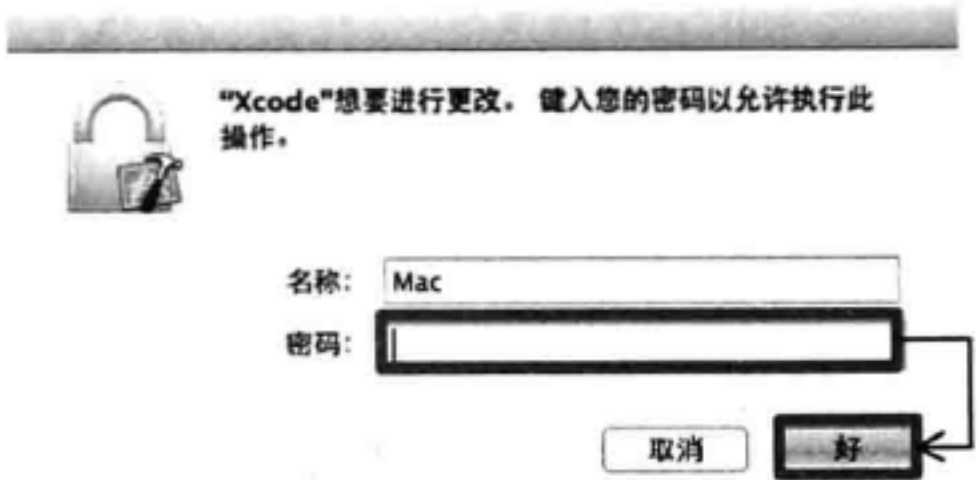


图 1.24 操作步骤 10

(10) 输入密码，单击“好”按钮，进行组件的安装，组件安装完成后，就会弹出 Welcome to Xcode 对话框。此时 Xcode 就被启动了，如图 1.25 所示。

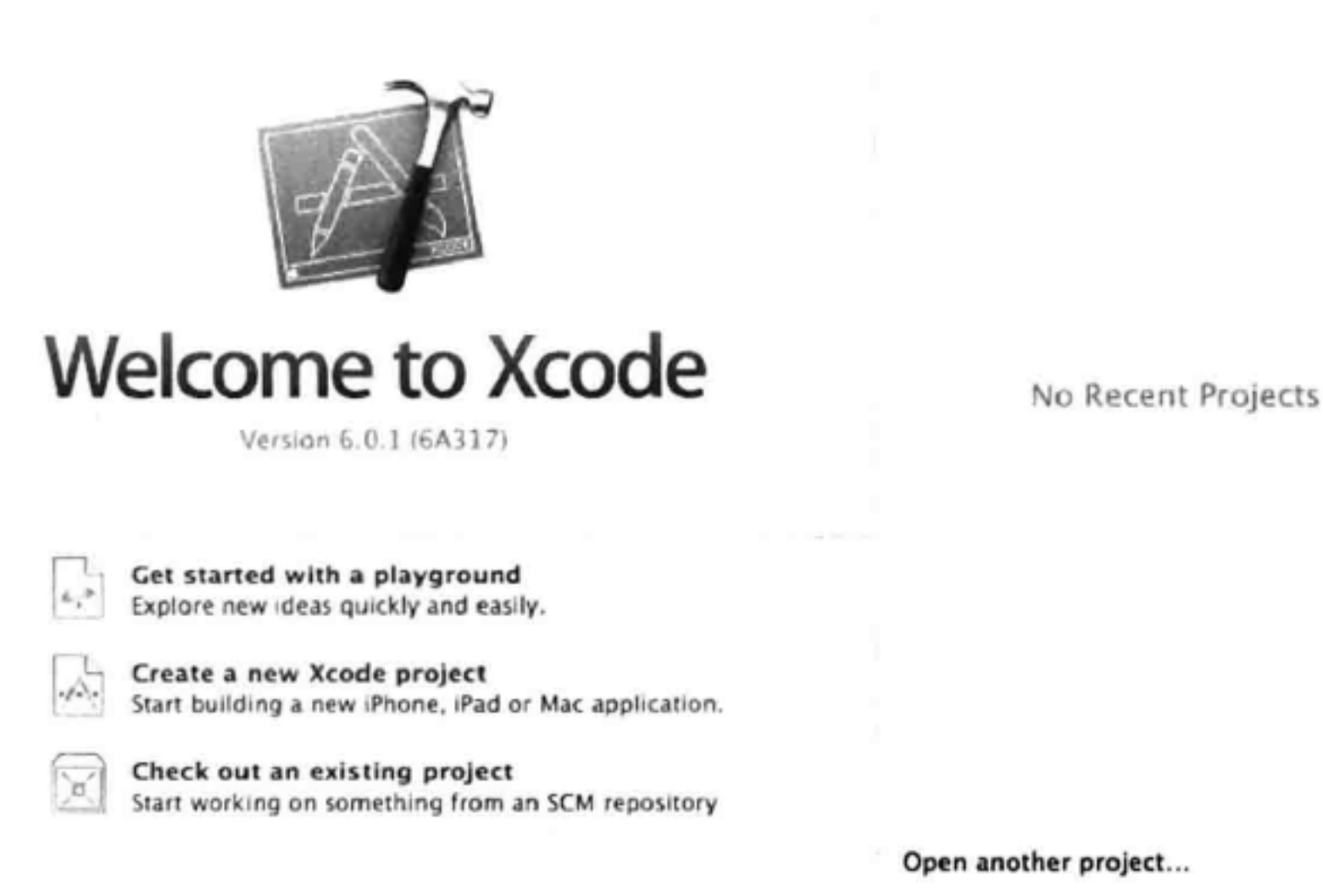


图 1.25 操作步骤 11

1.2.2 其他网站下载和安装 Xcode

有时候，应用商店下载较慢，所以用户也可以选择从其他网站下载 Xcode 安装文件。下面讲解这种 Xcode 的安装步骤。

(1) 双击下载的 Xcode 软件，弹出正在打开此软件的对话框，如图 1.26 所示。

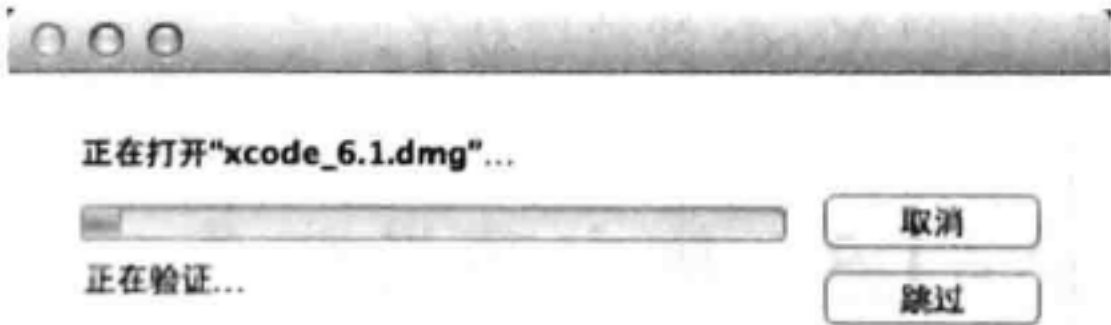


图 1.26 操作步骤 1

(2) 打开该软件后，就会弹出 Xcode 对话框，如图 1.27 所示。



图 1.27 操作步骤 2

(3) 将 Xcode 应用软件拖动到 Applications 文件夹中。此时该软件就会复制到应用程序中。

(4) 在菜单栏的“前往”|“应用程序”中找到安装的 Xcode，双击打开，弹出 Xcode and iOS SDK License Agreement 对话框，如图 1.28 所示。

(5) 单击 Agree 按钮，弹出“键入您的密码以允许执行此操作”对话框，如图 1.29 所示。

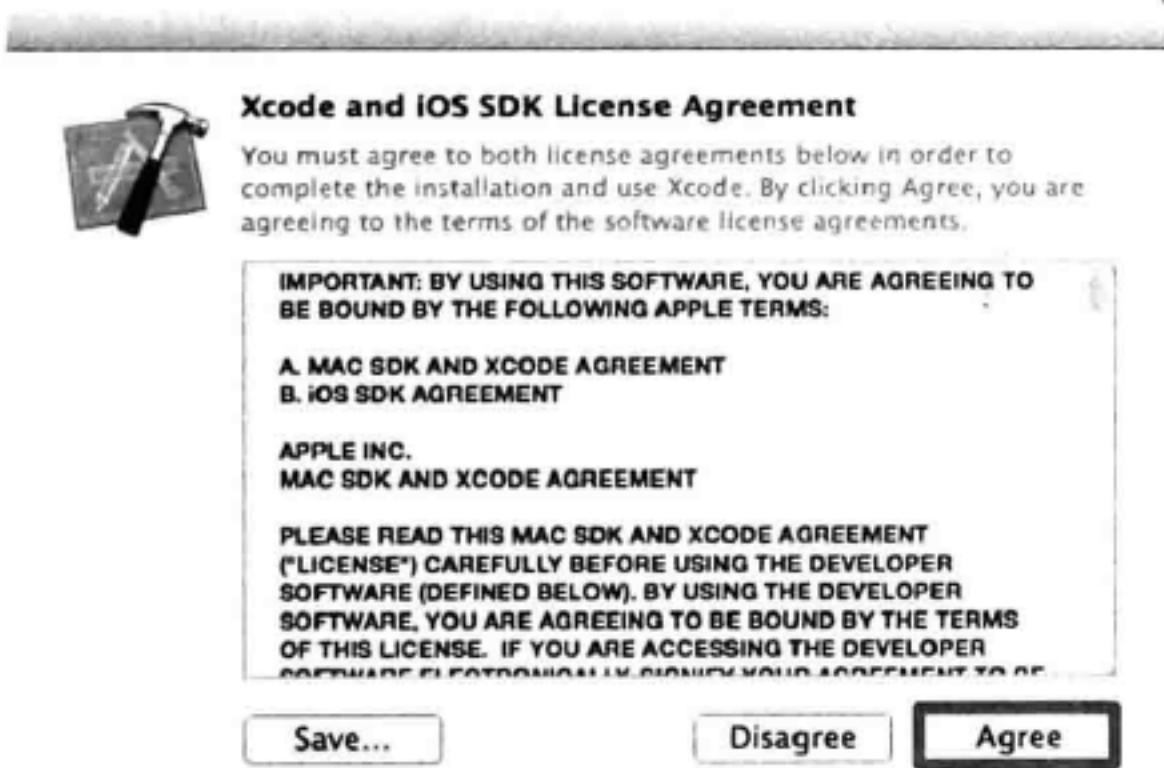


图 1.28 操作步骤 3

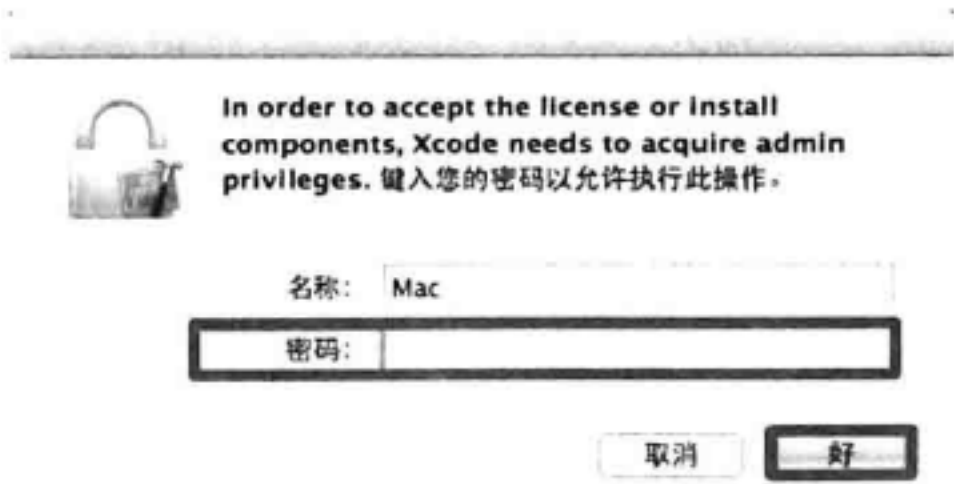


图 1.29 操作步骤 4

(6) 输入密码，单击“好”按钮，进行组件的安装，组件安装完成后，就会弹出 Welcome to Xcode 对话框，此时 Xcode 就被启动了。

1.3 绑定苹果账号

有时为了方便 Xcode 中组件以及内容的随时更新，必须要绑定一个苹果账号。以下将

讲解如何绑定一个苹果账号。

(1) 单击打开 Xcode，在菜单栏中选择 Xcode|Preferences 命令。

(2) 在弹出的对话框中选择 Accounts 选项，打开 Accounts 对话框，如图 1.30 所示。

(3) 选择“+”号，就会出现 3 个选项分别为 Add Apple ID...，Add Repository...和 Add Server...，如图 1.31 所示。

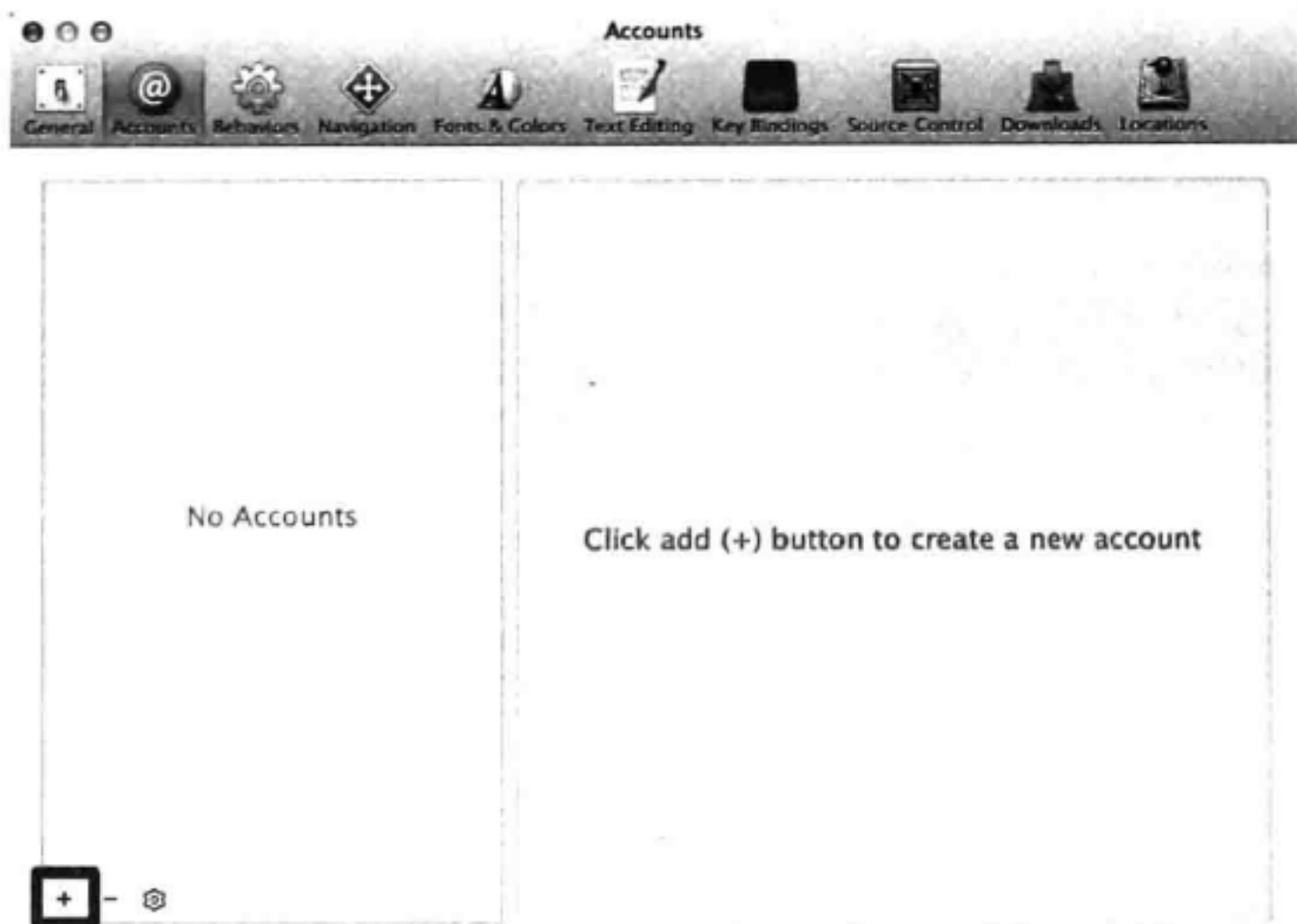


图 1.30 操作步骤 1

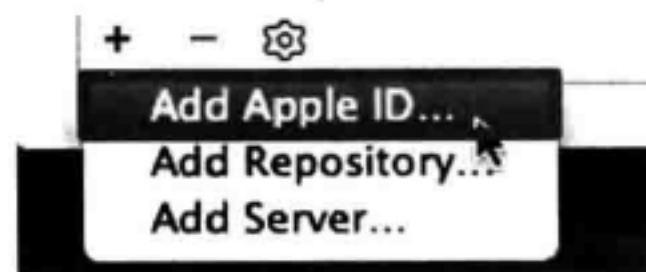


图 1.31 操作步骤 2

(4) 选择 Add Apple ID...命令，弹出一个 Enter an Apple ID associated with an Apple Developer Program:对话框，如图 1.32 所示。

(5) 输入苹果账号以及密码后，单击 Add 按钮，此时苹果账号就被绑定了，并将相关信息显示在 Accounts 对话框中，如图 1.33 所示。



图 1.32 操作步骤 3

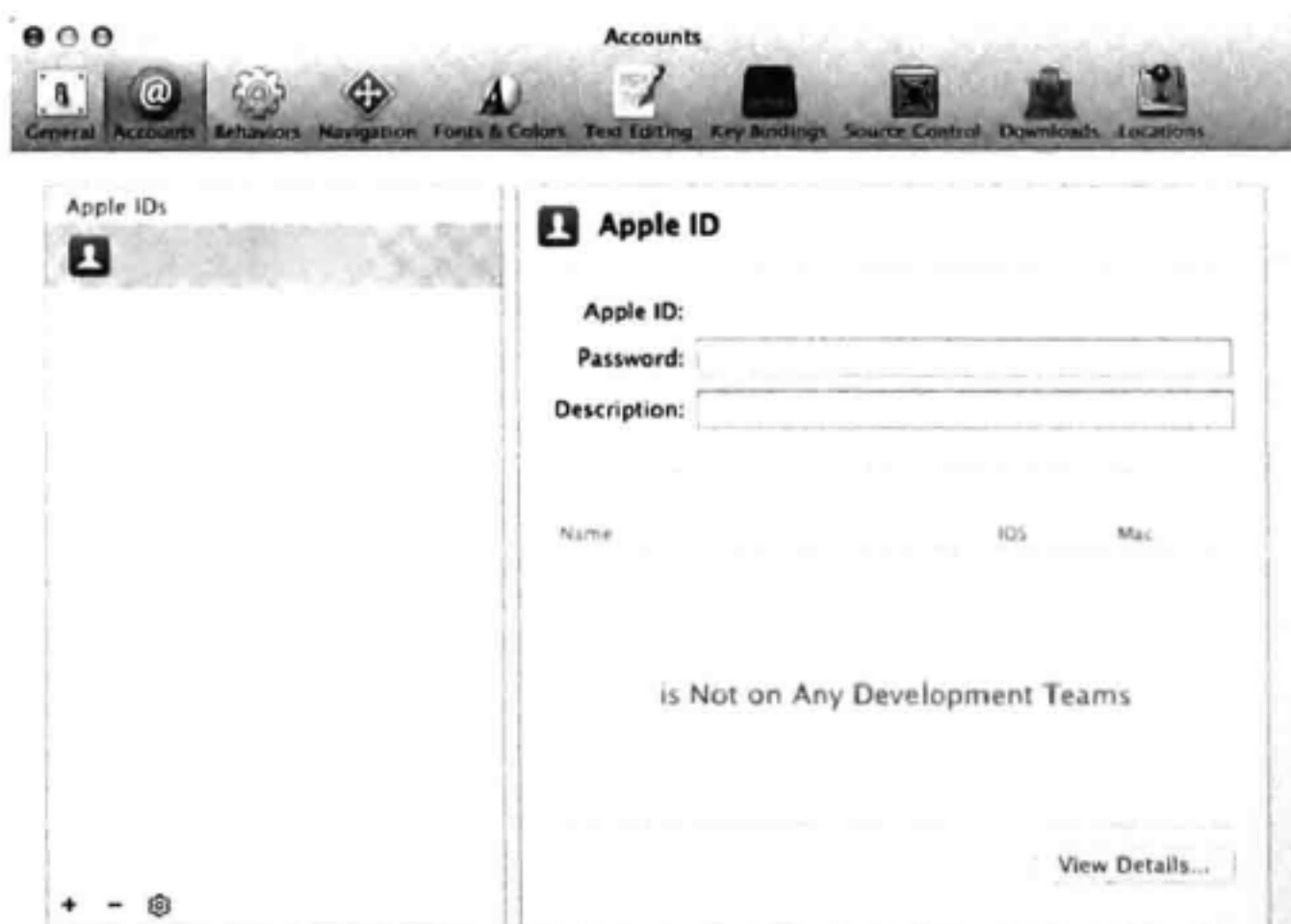


图 1.33 操作步骤 4

注意：在进行绑定苹果账号的操作时，Xcode 必须处于启动的状态。

1.4 更新组件和文档

Xcode 中组件和文档都是经常更新的。为了获得最新的工具和帮助文档，我们需要定时更新组件和文档。操作方式如下所述。

(1) 选择 Accounts 对话框中的 Downloads 选择，进入到 Downloads 对话框，如图 1.34 所示。

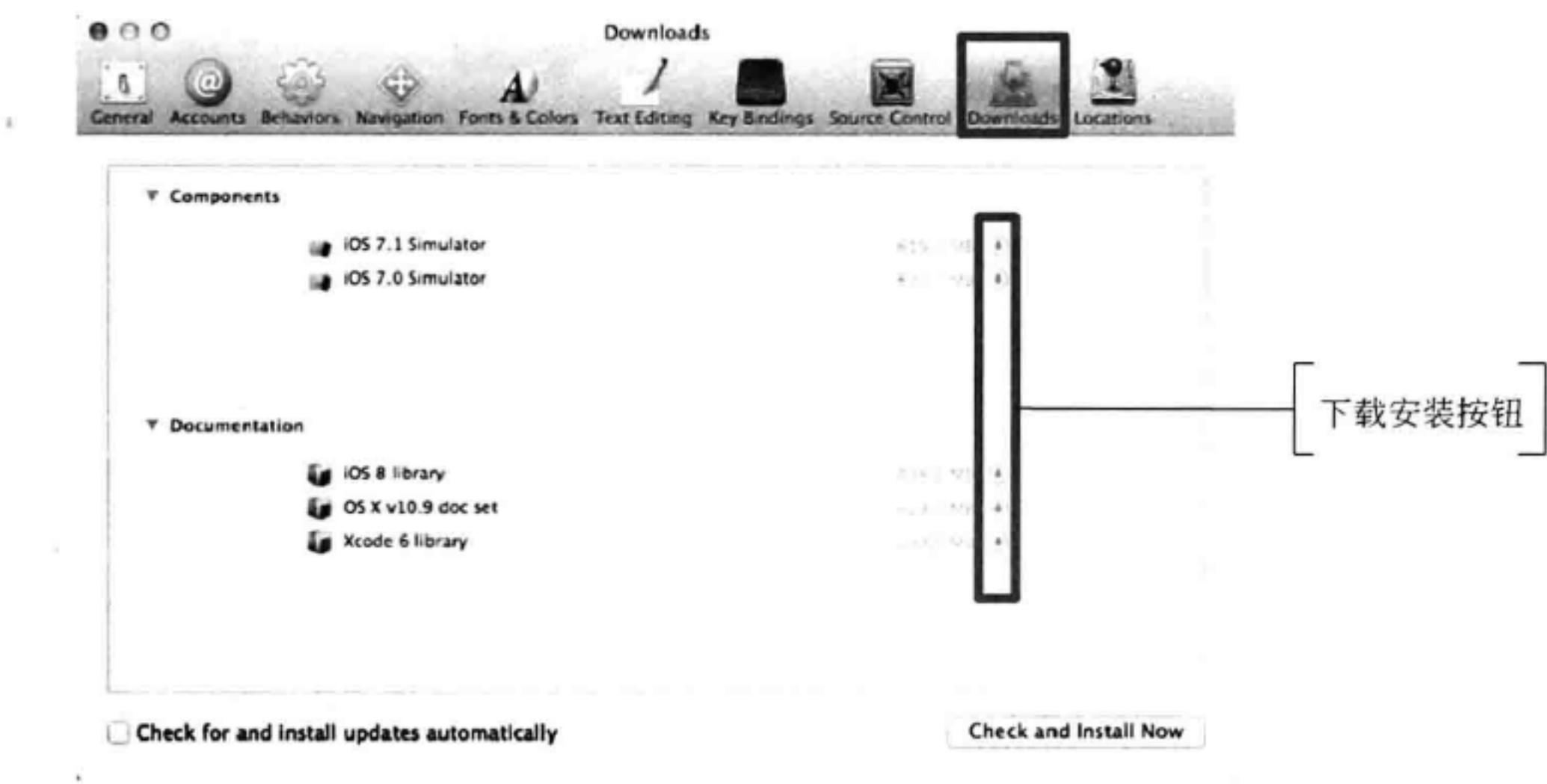


图 1.34 操作步骤 1

(2) 选择需要进行更新的文档，并单击组件后面的下载和安装按钮，进行组件和文档的下载和安装，如图 1.35 所示。



图 1.35 操作步骤 2

注意：在更新组件和文档的操作时，Xcode 必须处于启动的状态。

1.5 首次打开 Xcode

Xcode 安装好之后，就可以使用了。下面就讲解如何打开 Xcode，并实现一个应用程序项目的创建。一个 iOS 应用的所有文件都在 Xcode 项目下，项目可以帮助用户管理代码文件和资源文件。以下是打开 Xcode 并创建项目的具体操作步骤。

(1) 双击 Xcode 启动，进入 Welcome to Xcode 对话框中，如图 1.36 所示。

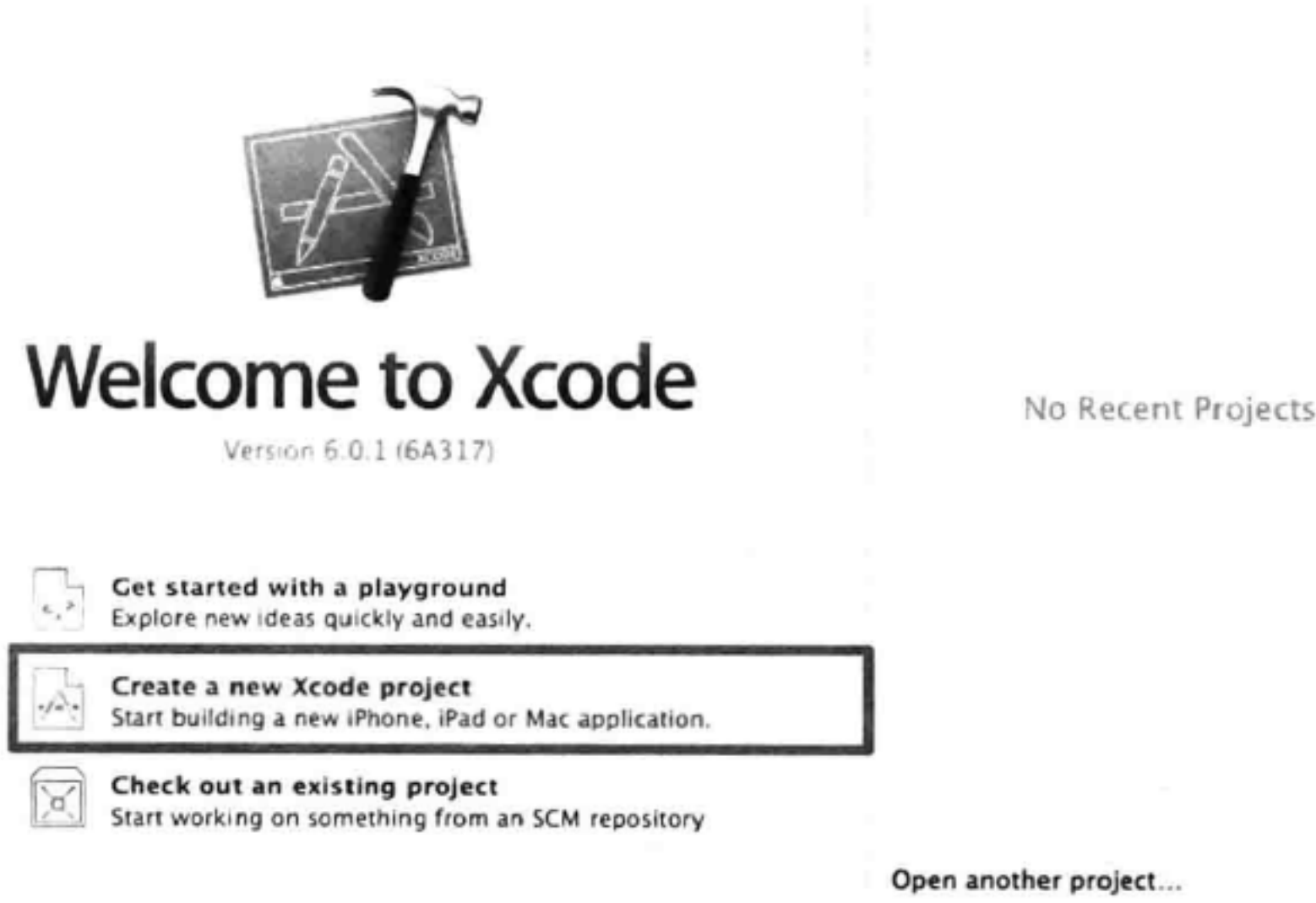


图 1.36 操作步骤 1

(2) 选择 Create a new Xcode project 选项，弹出 Choose a template for your new project: 对话框，如图 1.37 所示。

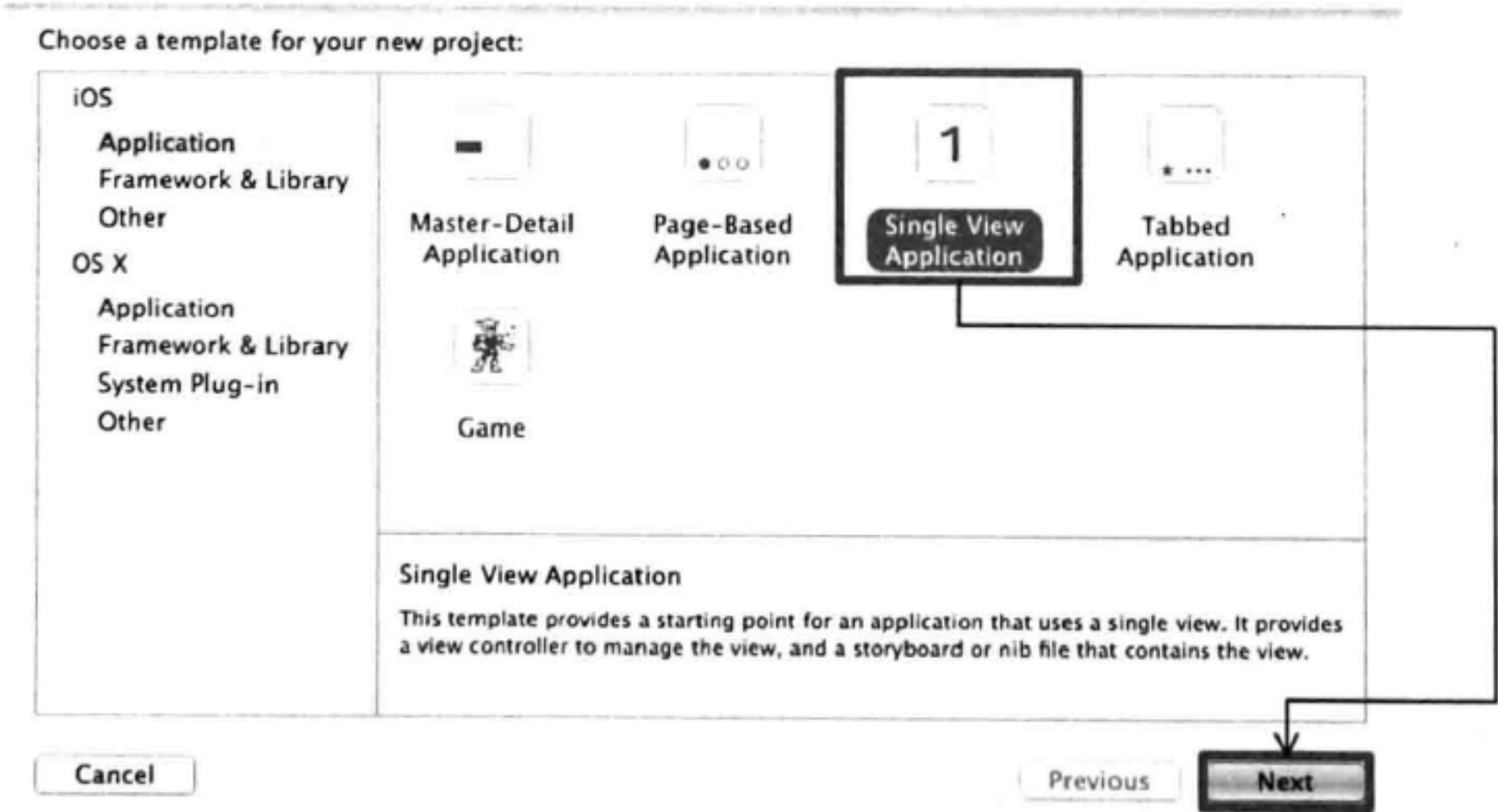


图 1.37 操作步骤 2

(3) 选择 iOS 下 Application 下的 Single View Application 模板，然后单击 Next 按钮，弹出 Choose options for your new project:对话框。在其中填入 Product Name（项目名）、Organization Identifier（标识符）信息，以及选择 Language（编程语言）和设备 Devices（设备），如图 1.38 所示。

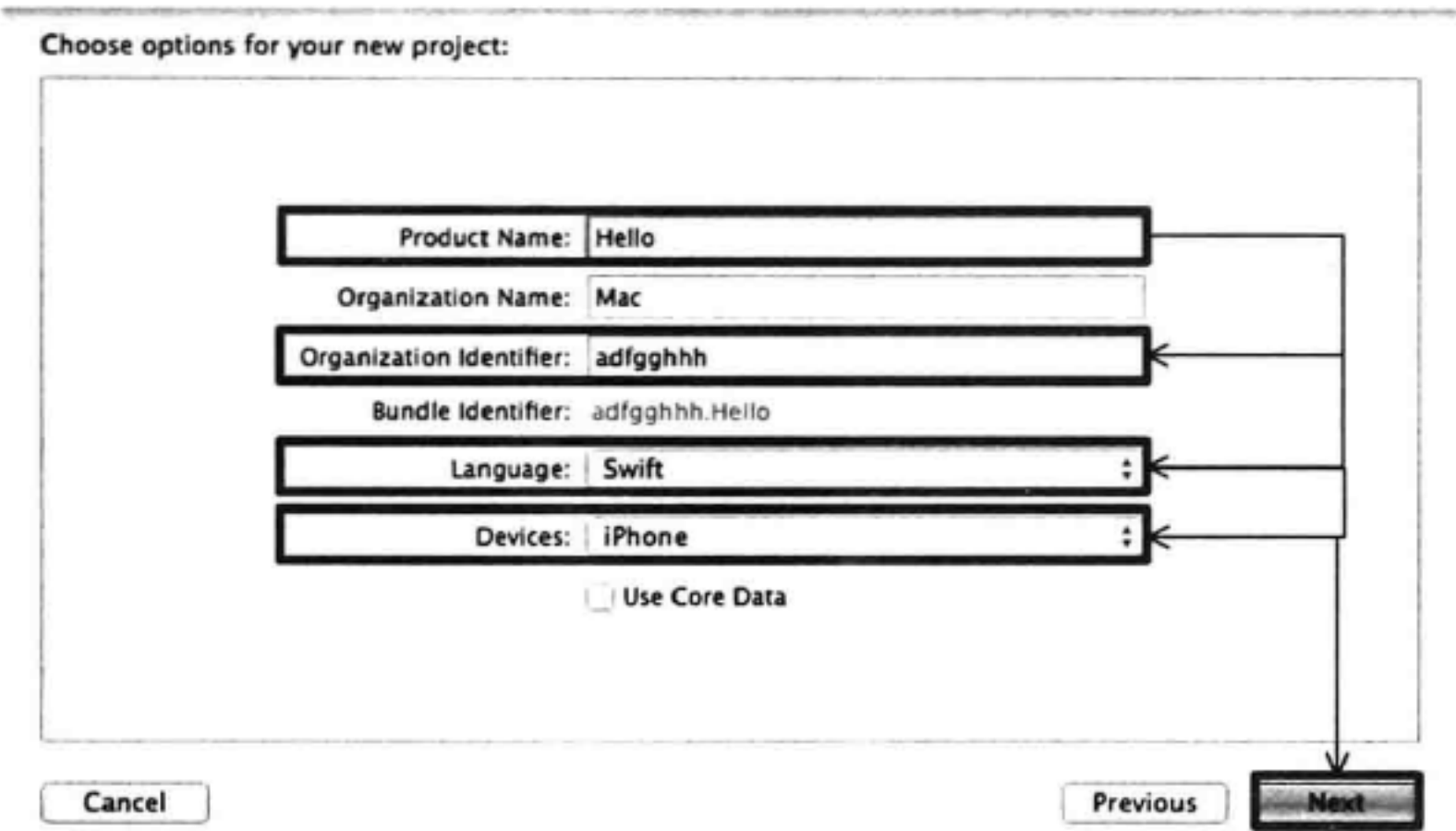


图 1.38 操作步骤 3

注意：Product Name 和 Organization Identifier 信息是随意的，由开发者决定。由于我们使用 Swift 语言编写程序，所以在 Language 这一项中选择 Swift。一般对 Organization Identifier、Language 和 Devices 设置一次后，在下一次创建项目时，到了 Choose options for your new project对话框时只需输入 Product Name(项目名)就可以了。

(4) 单击 Next 按钮，弹出保存位置对话框，如图 1.39 所示。



图 1.39 操作步骤 4

(5) 选择 Create 按钮，这时一个项目名为 Hello 的项目就创建好了。

1.6 Xcode 的界面介绍

一个 Xcode 项目由很多文件组成，例如代码文件和资源文件等。Xcode 会帮助开发者对这些文件进行管理。所以，Xcode 的界面也比较复杂，如图 1.40 所示。



图 1.40 Xcode 界面

在图中可以看到，Xcode 的界面大致可以分为 5 大部分。其中，编号为 1 的部分是导航窗口；编号为 2 的部分是代码编辑区域；编号为 3 的部分是工具窗口；编号为 4 的部分是显示程序调试信息的窗口；编号为 5 的部分是工具窗口。本节将对几个重要的区域进行讲解。

1.6.1 导航窗口

导航窗口的作用是显示整个项目的树状结构。开发者可以根据自己的喜好对其进行大小调整，以及显示和隐藏（通过 View|Navigators|Show/HideNavigator 命令来实现显示和隐藏，或通过使用 Hide or show the Navigator 按钮来实现显示和隐藏）。导航窗口可以显示 8 类不同的信息，所以又有了 8 个导航器。这 8 个导航器分别为项目导航器、符号导航器、搜索导航器、问题导航器、测试导航器、调试导航器、断点导航器和日志导航器。可以通过导航窗口顶部的 8 个图标来进行导航之间的切换，如图 1.41 所示。

1.6.2 工具窗口

工具窗口可以对项目的一些设置信息进行编辑，开发者也可以对项目的信息进行随时

显示和隐藏（通过 View|Utilities|Show/HideUtilities 命令来实现显示和隐藏，通过使用 Hide or show the Utilities 按钮来实现显示和隐藏）。工具窗口可以分为上、下两个部分。上半部分显示的内容取决于开发者在编辑器上正在编辑的文件类型，其中文件类型有如下两种：

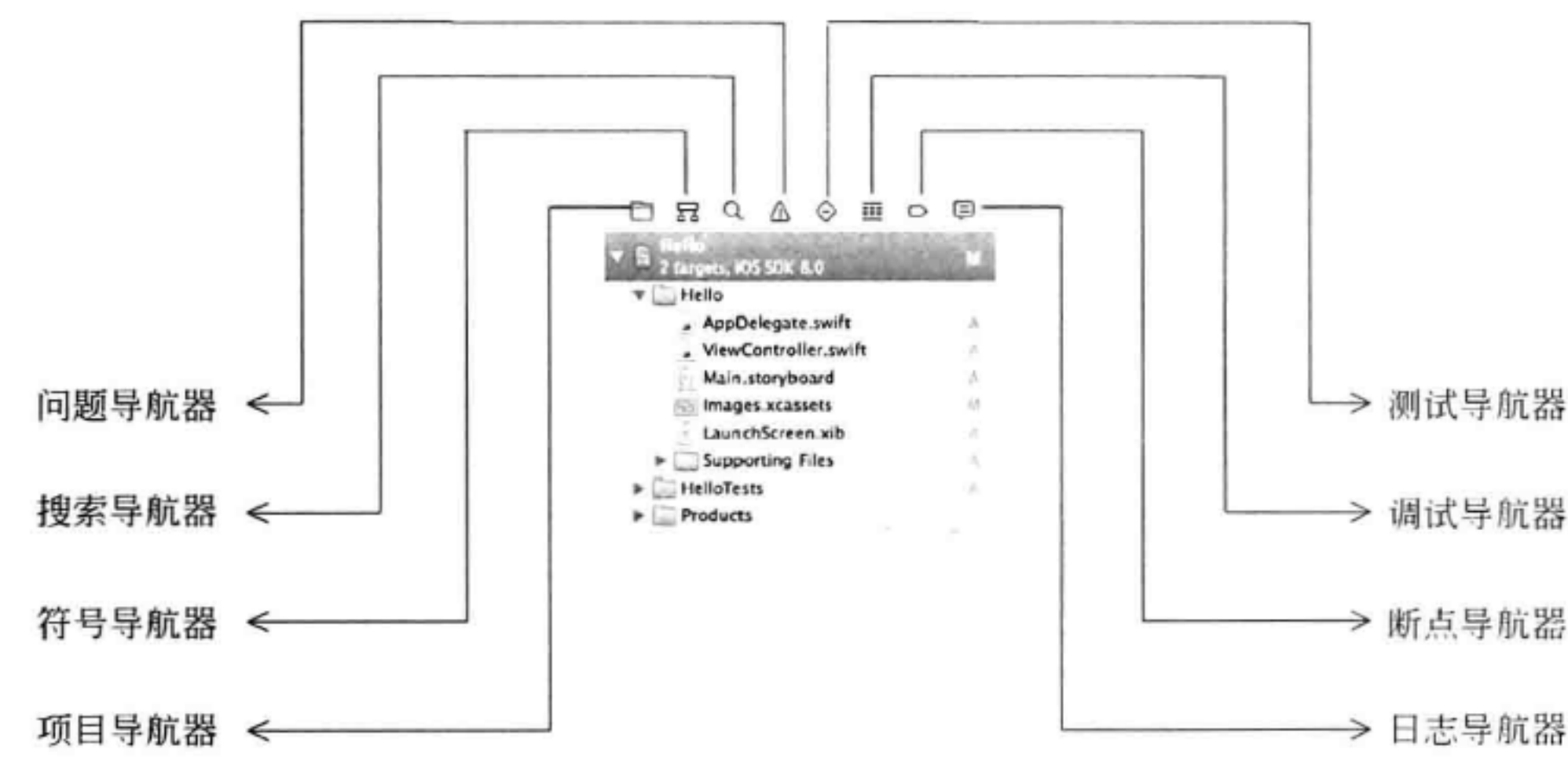


图 1.41 导航窗口

- ❑ 当编辑器编辑的是代码文件时，工具窗口上半部分显示的内容为文件查看器和快速帮助的其中一个的内容，如图 1.42 所示。要想实现两个内容的切换，可以通过使用此窗口上半部分在顶部显示的图标来进行切换。
- ❑ 当编辑器编辑的是界面文件时，工具窗口上半部分显示的内容为文件查看器、快速帮助、标识查看器、属性查看器、尺寸查看器和连接查看器的其中一个的内容，如图 1.43 所示。要想实现这 6 个内容的切换，可以通过使用此窗口上半部分在顶部显示的图标来进行切换。



图 1.42 编辑代码显示的内容

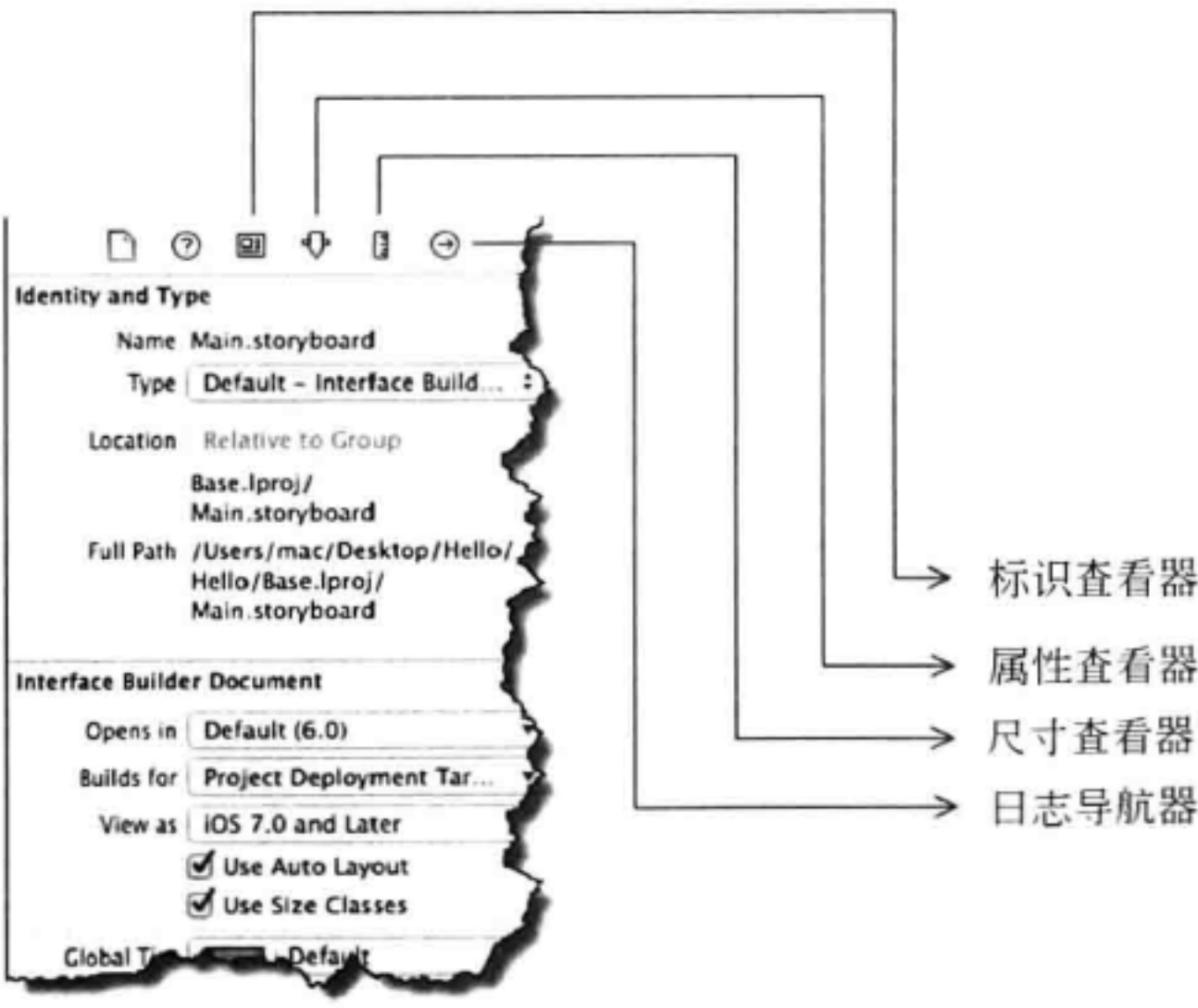


图 1.43 编辑界面显示的内容

上半个工具窗口介绍完后，再来看下半个工具窗口。下半个工具窗口显示的内容是文

件模板库、代码片断库、对象库和媒体库的其中一个内容，如图 1.44 所示。要想实现这 4 个内容的切换，可以通过使用此窗口下半部分在顶部显示的图标来进行切换。

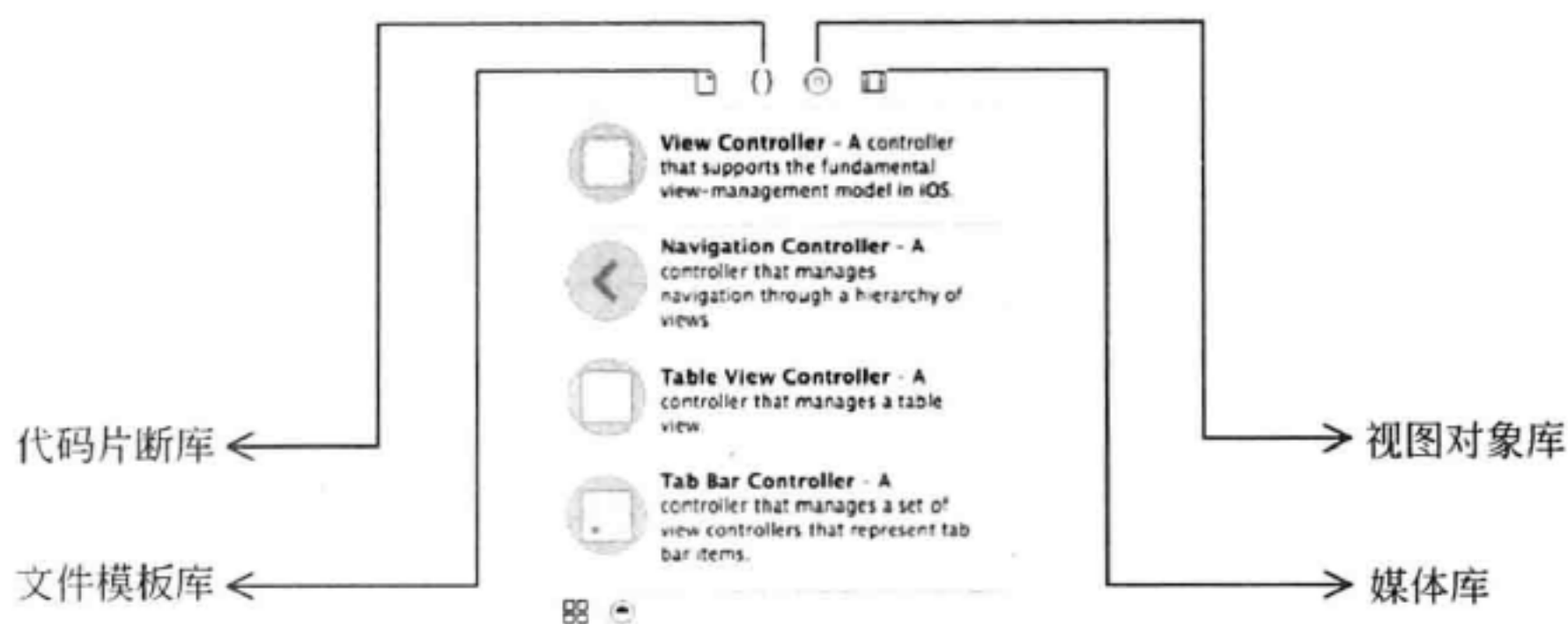


图 1.44 工具窗口的下半部分

1.6.3 编辑窗口

编辑窗口可以用来编写代码，或在编辑界面设置用户界面。在顶端，有左右箭头和整个项目的层次显示。编辑窗口可以记住已经编辑的文件名，可以使用顶端的左右箭头来选择曾经编辑过的文件。在一个项目的窗口中至少要包含一个编辑窗口。如果项目要同时打开多个编辑窗口该如何实现呢？以下将主要讲解 3 个方法。

1. Assistants方法

单击窗口中的 Show the Assistant editor 按钮后，Xcode 会默认显示两个编辑窗口（后面的窗口叫做 Assistant）。这两个窗口上的内容一般都是关联的。如果需要显示更多的编辑窗口，可以在 Assistant 窗口的顶部单击“+”按钮来实现。

2. Tabs方法

Tabs 方法是显示各个文件，和 Safari 显示网页的方法一样。通过选择 File|New|Tab 命令或使用快捷键 Windows+T 来启动这个方法。通过单击标签或者使用快捷键 Windows+Shift+} 在不同的窗口中进行切换。

3. 新的窗口

新的窗口这个方法和 Tab 类似，但是它是用来显示独立的窗口。要创建新的窗口可以选择 File|New|Window 命令来创建。

1.6.4 目标窗口

目标窗口中包含了项目的程序和配置，这些配置指定了如何构建程序代码，如图 1.45 所示。在目标窗口的顶部，可以选择 General、Capabilities、Info、Build Settings、Build Phases

和 Build Rules 中的内容。

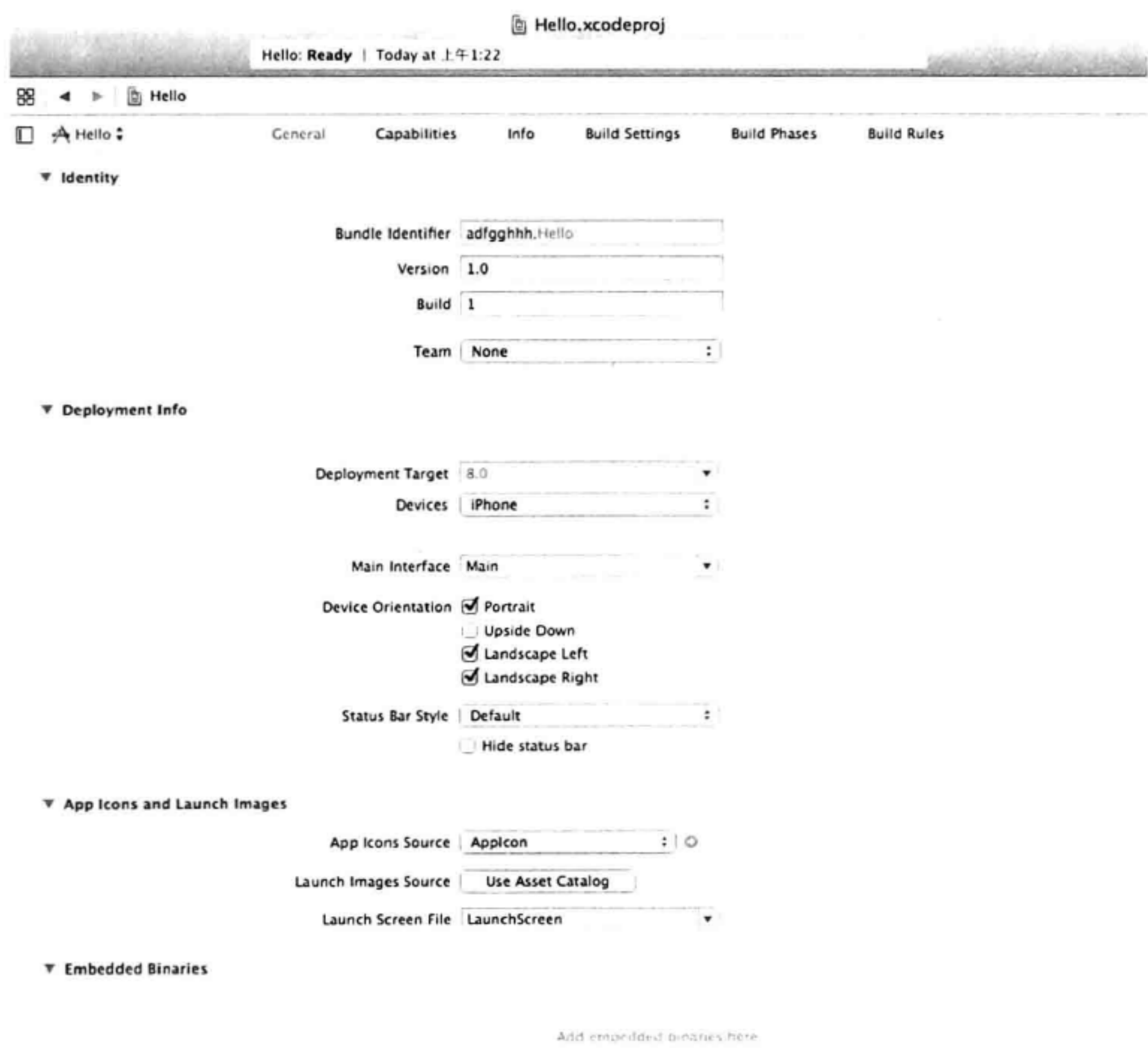


图 1.45 目标窗口

第 2 章 编写第一个 Swift 程序

Swift 语言是 2014 年 6 月苹果公司在开发者大会上发布的用于开发 iOS 应用程序的语言。本章将使用 Swift 语言开发一个简单的 iOS 应用程序，并且对 iOS 模拟器、编辑界面，以及调试、真机测试等进行详细的讲解。

2.1 运行程序

在第 1 章创建好 Hello 项目之后，就可以在此项目中单击如图 2.1 所示的运行按钮运行程序了。在运行程序前首先对程序进行编译，如果程序正确，会出现一个 Build Succeeded 提示，如图 2.2 所示。如果程序出现错误，那么就会出现一个 Build Failed 提示，如图 2.3 所示。



图 2.1 运行按钮



图 2.2 编辑成功



图 2.3 编译失败

在程序编译后，会自动对程序进行连接和运行，运行效果如图 2.4 和图 2.5 所示。



图 2.4 运行效果

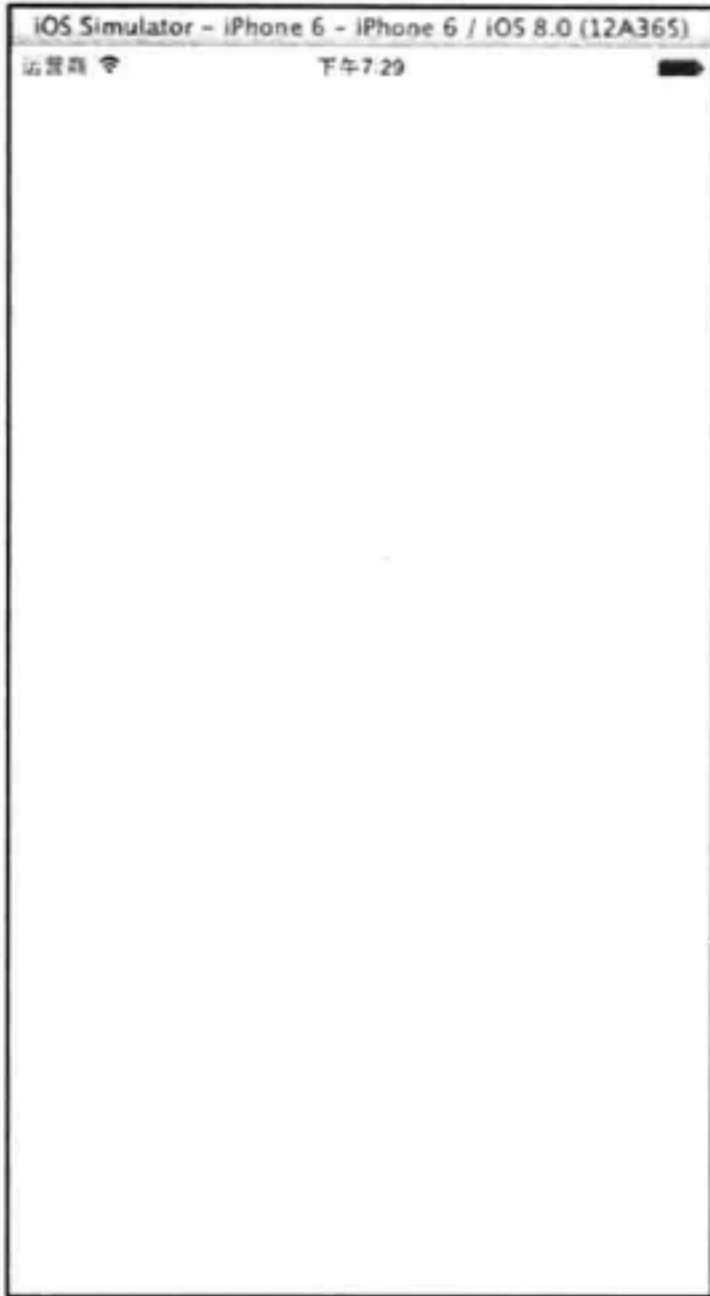


图 2.5 运行效果


 **注意：**由于没有对程序进行编写，也没有对编辑界面进行设置，所以这时的运行效果是会产生任何效果的。对于编辑界面会在后面做一个详细的介绍。

图 2.4 是应用程序的一个启动界面，是系统自带的，开发者真正要使用到的界面是图 2.5 所示的界面。启动界面也是可以删除的，如果开发者不想在程序运行时有启动界面，可以打开 Info.plist 文件，在此文件中找到 Launch screen interface file base name，将其 value 后面的内容删除，如图 2.6 所示。

Key	Type	Value
▼ Information Property List	Dictionary	(14 items)
Localization native development r...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	adfgghh.\$(PRODUCT_NAME:rfc1034identifier)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone envir...	Boolean	YES
Launch screen interface file base...	String	
Main storyboard file base name	String	Main
► Required device capabilities	Array	(1 item)
► Supported interface orientations	Array	(3 items)

图 2.6 去除启动界面

此时再运行程序，就可以直接看到 2.5 所示的运行效果了。

2.2 模拟器的操作

从图 2.4 和图 2.5 所看到的类似于手机的模型就是 iOS 模拟器。iOS 模拟器是在没有 iPhone 或 iPad 设备时，对程序进行检测的设备。iOS 模拟器可以模仿真实的 iPhone 或 iPad 等设备的各种功能。本节将讲解一些有关模拟器的操作。

2.2.1 模拟器与真机的区别

iOS 模拟器可以模仿真实的 iPhone 或 iPad 等设备的各种功能，如表 2-1 所示。

表 2-1 iOS 模拟器

方 面	功 能
旋转屏幕	向上旋转
	向下旋转
	向右向左旋转
手势支持	轻拍
	触摸与按下
	轻拍两次
	猛击
	轻弹
	拖动
	捏

iOS 模拟器只能实现表 2-1 中的这些功能，其他的功能是实现不了的，如打电话、发送 SMS 信息、获取位置数据、照照相和麦克风等。

2.2.2 退出应用程序

如果想要将图 2.5 所示的应用程序退出（为用户完成某种特定功能所设计的程序被称为应用程序），该怎么办呢？这时就需要选择菜单栏中的 Hardware|Home 命令，退出应用程序后的效果如图 2.7 所示。

2.2.3 应用程序图标的位置

在图 2.7 中可以看到应用程序的图标是网状白色图像，它是 iOS 模拟器上的应用程序默认的图标。这个图标是可以进行改变的。以下就来实现现在 iOS 模拟器上将 Hello 应用程序的图标进行更改。

(1)添加图像 logo.png 到创建的项目中，添加图像的具体步骤如下所述。右击 Supporting Files 文件夹，弹出快捷菜单，如图 2.8 所示。

(2) 选择 Add Files to "Hello"...命令，弹出选择文件对话框，如图 2.9 所示。

最后，选择需要添加的图像，单击 Add 按钮，实现图像的添加。添加后的图像就会显示在 Supporting Files 文件夹中。



图 2.7 退出应用程序

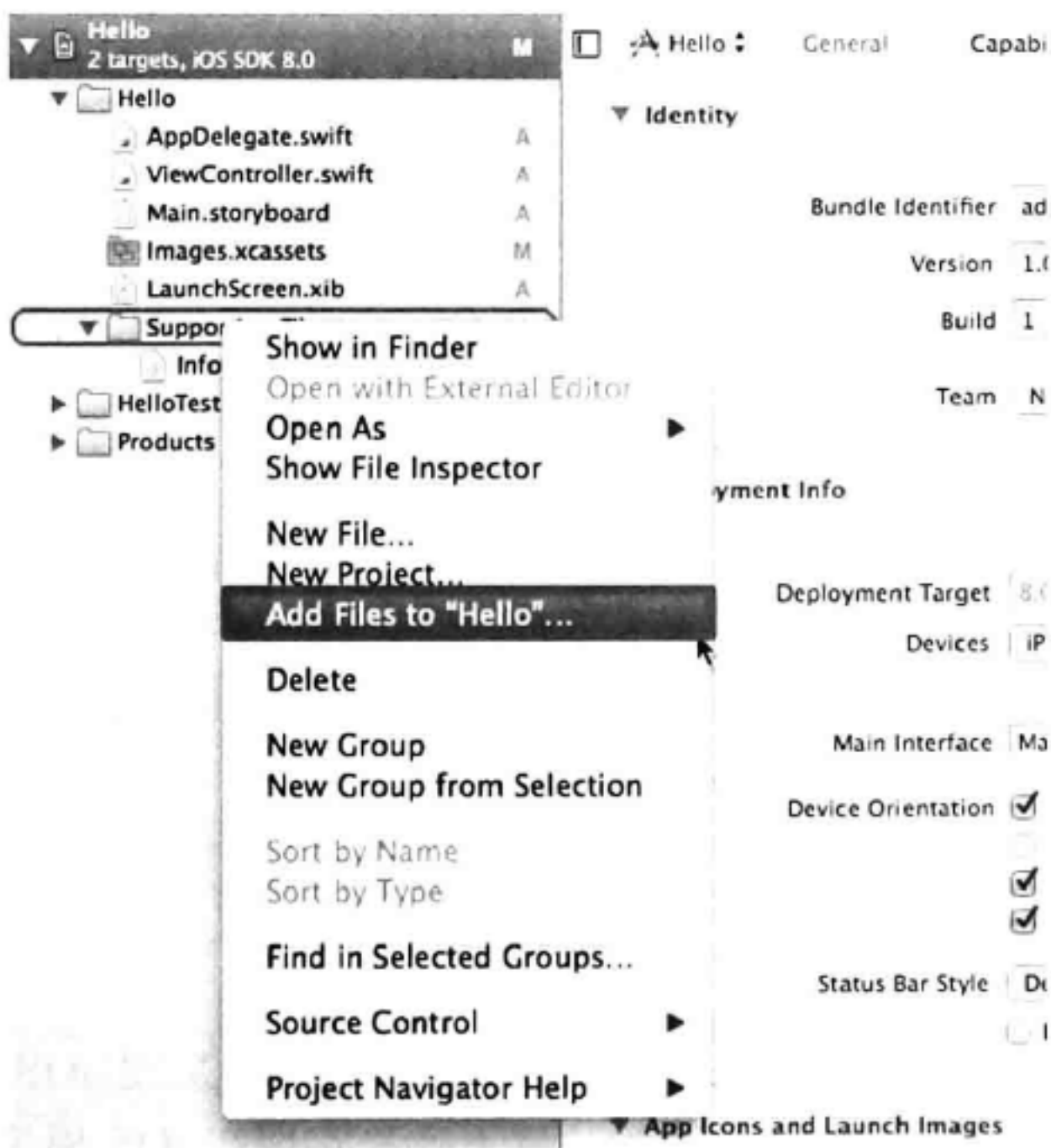


图 2.8 添加图像 1

(3) 单击打开 Supporting Files 文件夹中的 Info.plist 文件，在其中添加一项 Icon files，在其下拉菜单的 Value 中输入添加到 Supporting Files 文件夹中的图片的名称，如图 2.10 所示。

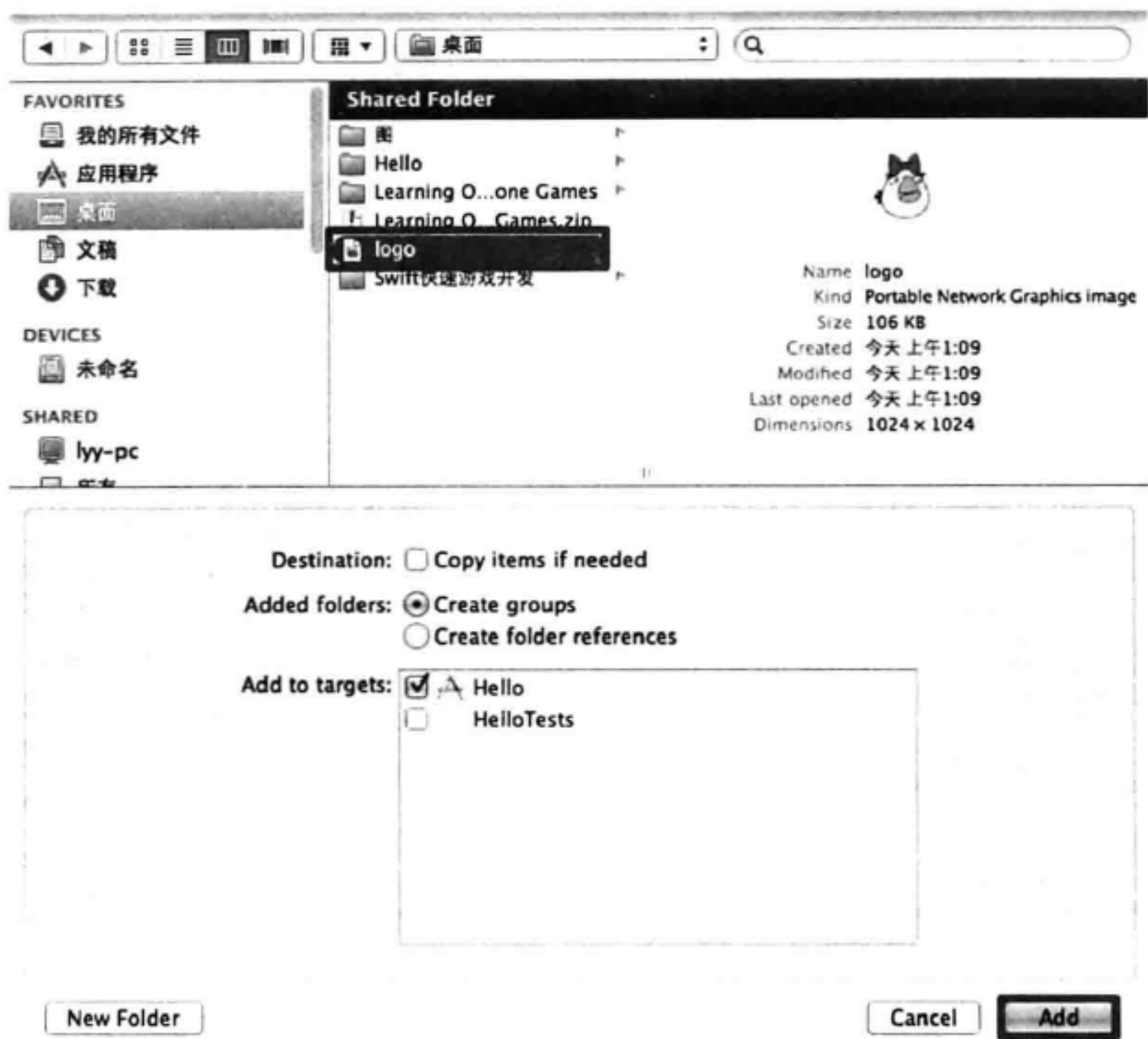


图 2.9 添加图像

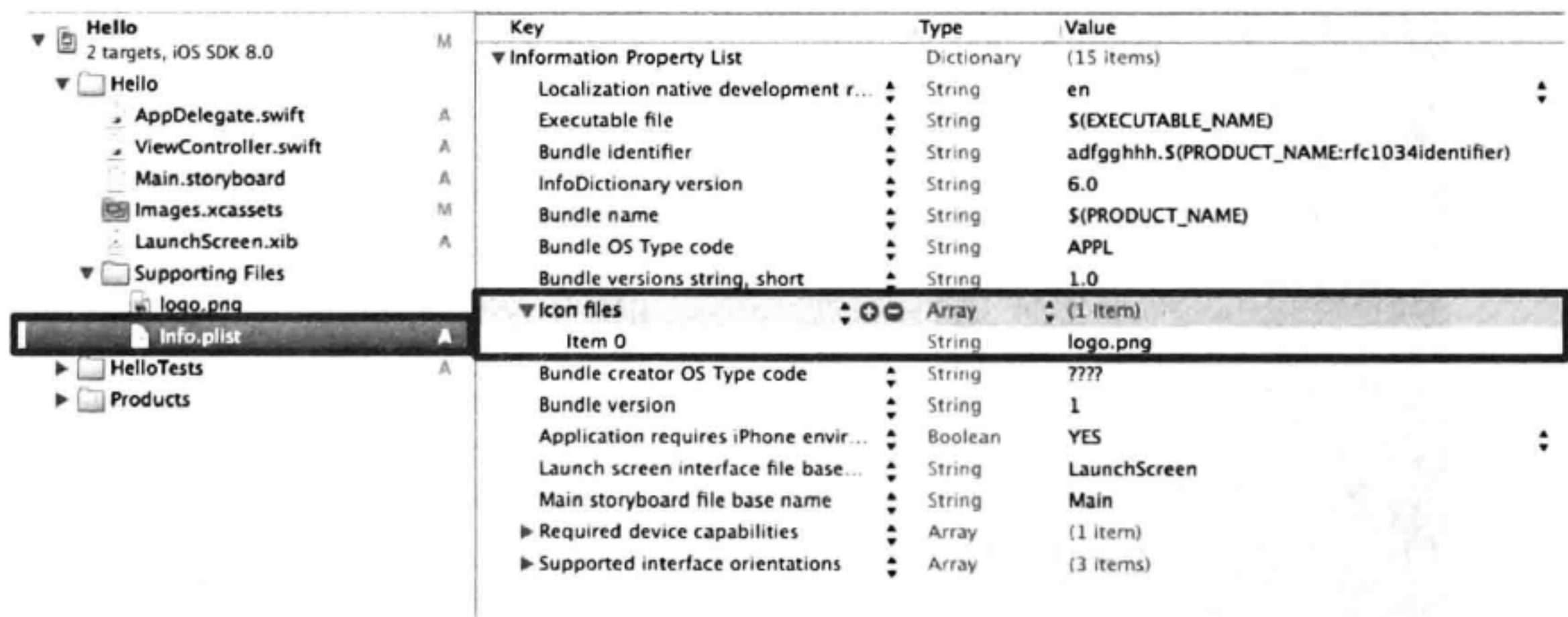


图 2.10 设定图标

此时运行程序，会看到如图 2.11 所示的效果。

2.2.4 语言设置

一般情况下，iOS 模拟器默认使用的 English（英语），对于英文不好的开发者来说，英文就像天书，怎么看也看不懂。这时，就需要将 iOS 模拟器的语言进行设置。要设置语言，需要切换到模拟器的主界面，向左拖动，找到 Settings 应用程序。找到后就可以对 iOS 模拟器的语言进行设置了，以下将 iOS 模拟器的语言变为中文，具体操作步骤如下所述。



图 2.11 运行效果

- (1) 切换到主界面，找到 Settings 应用程序，如图 2.12 所示。
- (2) 选择 Settings 应用程序图标，进入 Settings 界面中，如图 2.13 所示。



图 2.12 操作步骤 1

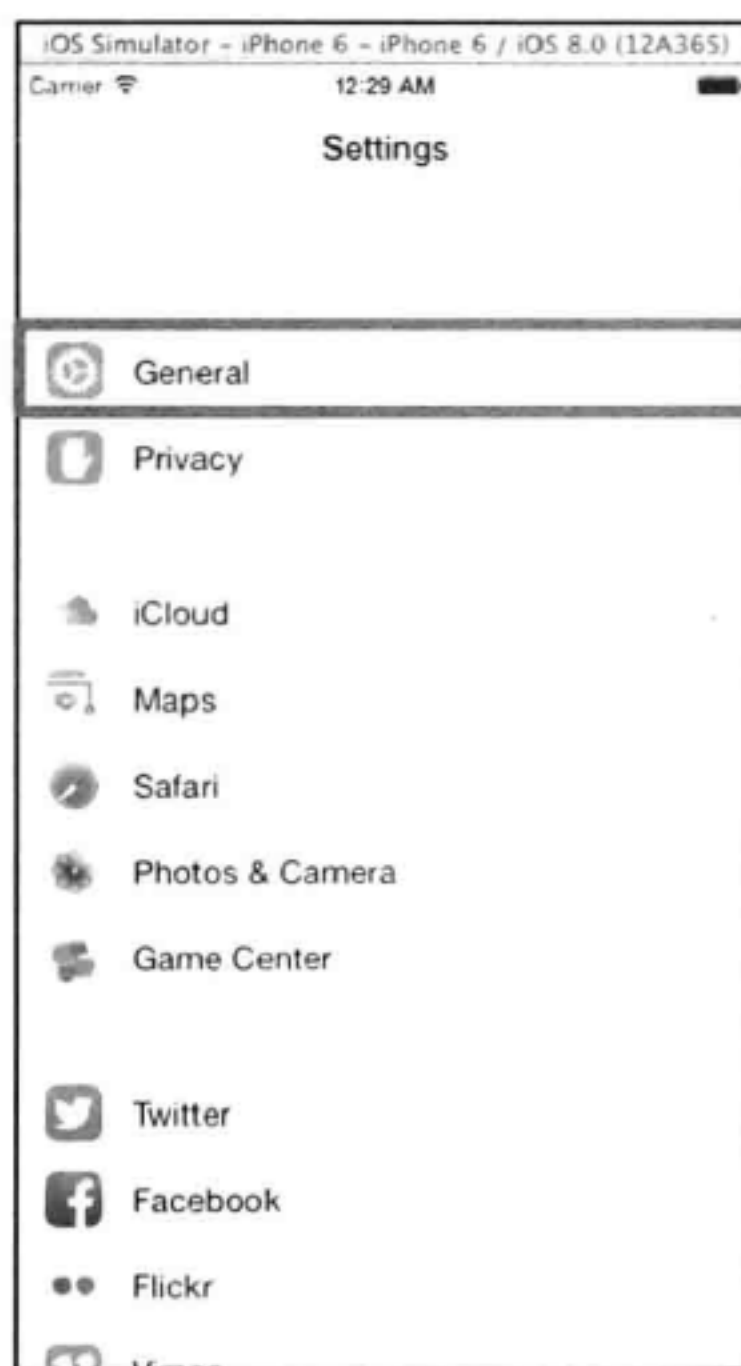


图 2.13 操作步骤 2

- (3) 选择 General 选项，进入 General 界面，如图 2.14 所示。
- (4) 选择 Language&Region 选项，进入 Language&Region 界面中，如图 2.15 所示。

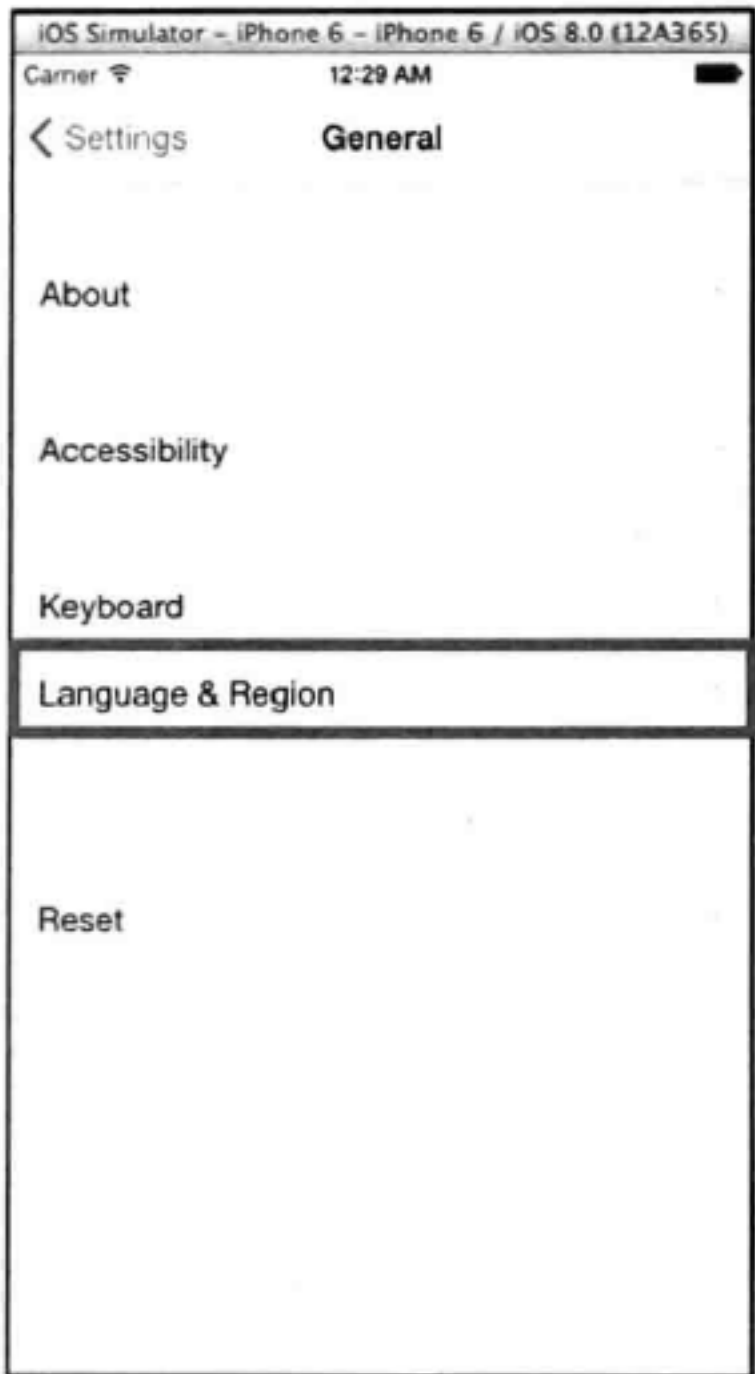


图 2.14 操作步骤 3



图 2.15 操作步骤 4

- (5) 选择 iPhone Language 选项，进入 iPhone Language 界面，如图 2.16 所示。
- (6) 选择“简体中文”选项，轻拍 Done 按钮，弹出动作表单，如图 2.17 所示。



图 2.16 操作步骤 5



图 2.17 操作步骤 6

(7) 选择 Change to Chinese, Simplified 选项，进入正在设置语言的界面，如图 2.18 所示。当语言设置好后，iOS 模拟器将会退出到主界面，此时主界面的应用程序的标题名

就变为了中文，如图 2.19 所示。



图 2.18 操作步骤 7



图 2.19 操作步骤 8

2.2.5 旋转

在前面介绍过 iOS 模拟器可以模仿真实的 iPhone 或 iPad 等设备的屏幕旋转（上、下、左、右），以下就来使用手动的方式将 iOS 模拟器进行旋转。要实现 iOS 模拟器的旋转，只需要同时按住 Command+方向键就可以了。以下是使用 Command+右键实现的将 iOS 模拟器进行向右旋转，如图 2.20 所示。

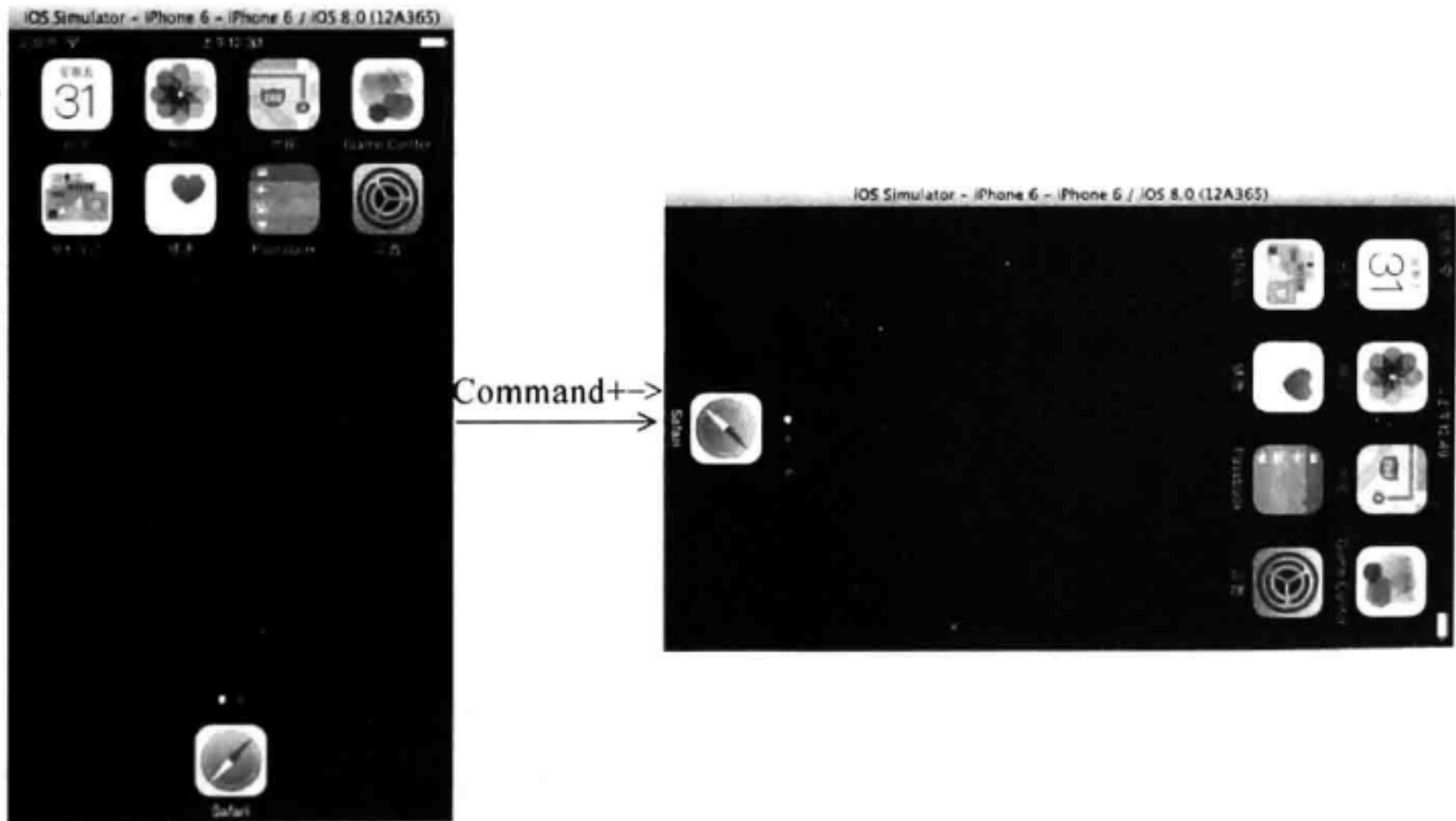


图 2.20 旋转 iOS 模拟器

2.2.6 删除应用程序

如果在 iOS 模拟器中出现了很多的应用程序,就可以将不再使用的应用程序进行删除。这样一来可以为设备节省内存空间,也可以使用户或者开发者便于管理自己的应用程序。以下主要实现 Hello 应用程序的删除。

(1) 长按要删除的 Hello 应用程序,直到所有的应用程序都开始抖动,并在每一个应用程序的左上角出现一个“x”,它是一个删除标记,如图 2.21 所示。

(2) 单击 Hello 程序左上角出现的删除标记,会弹出一个“删除"Hello World"”对话框,选择其中的“删除”按钮,如图 2.22 所示。这时 Hello 应用程序就在 iOS 模拟器上被删除了。



图 2.21 删除应用程序 1



图 2.22 删除应用程序 2

2.3 编辑界面

在 2.1 节中提到过编辑界面 (Interface Builder), 编辑界面是用来设计用户界面的, 单击打开 Main.storyboard 文件就可以打开编辑界面。在 Xcode 6.0.1 中, 编辑界面直接使用的是故事面板。本节将对编辑界面进行介绍。

2.3.1 界面介绍

单击 Main.storyboard 打开编辑界面后, 可以看到编辑界面由 4 部分组成, 如图 2.23

所示。

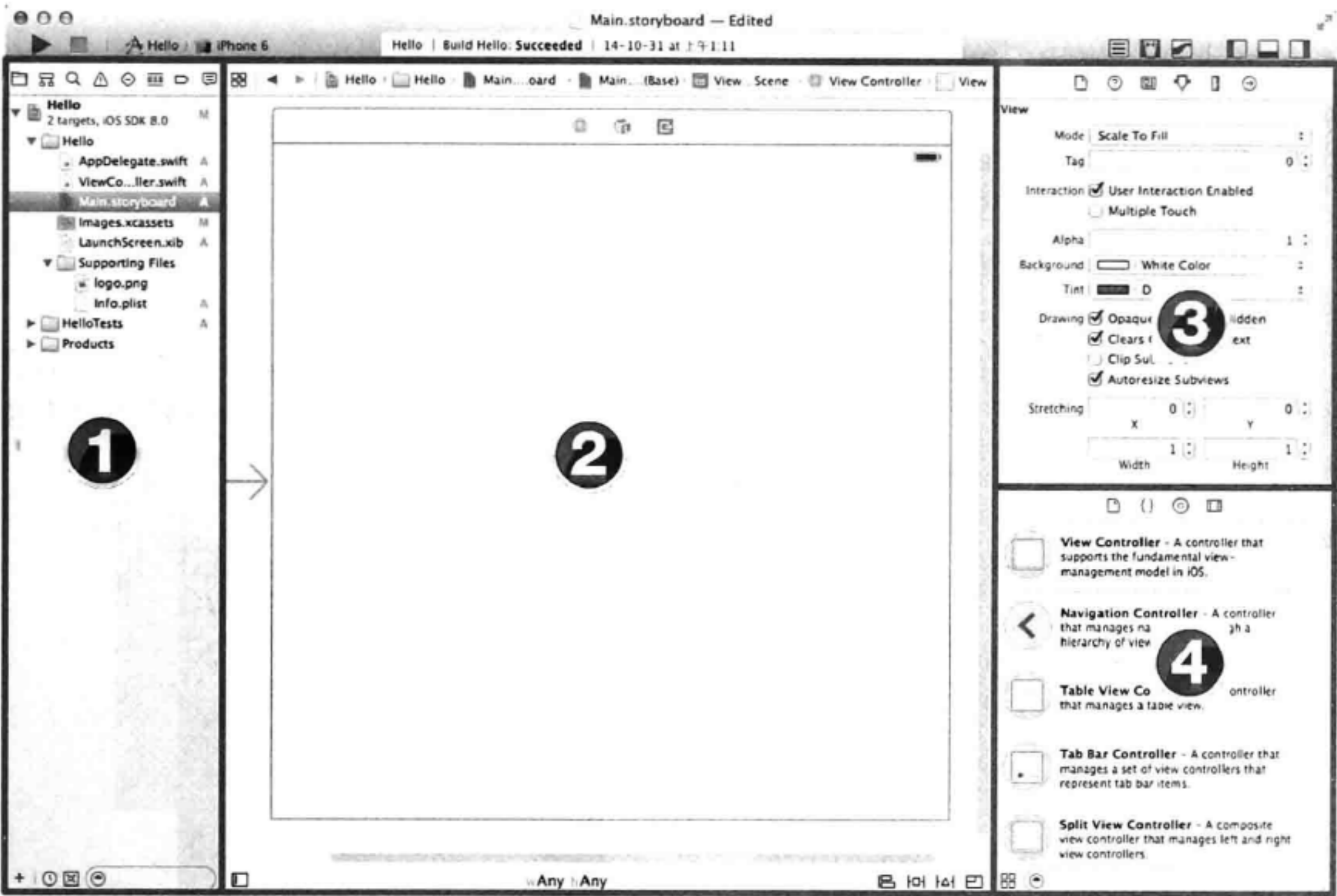


图 2.23 编辑界面

其中，这 4 部分的说明如下。

- ❑ 编号为 1 的部分为导航窗口。
- ❑ 编号为 2 的部分为画布，用于设计用户界面的地方，在画布中用箭头指向的区域就是界面，在画布中可以有多个界面，一般将界面称为场景或者说是一个视图。
- ❑ 编号为 3 的部分为工具窗格的检查器，用于编辑当前选择的对象的属性。
- ❑ 编号为 4 的部分为工具窗格的库，如果选择的是 Objects，里边存放了很多的视图。

在画布的界面上方有一个小的 Dock，它是一个文件管理器的缩写版。

Dock 展示场景中第一级的控件，每个场景至少有一个 View Controller、一个 First Responder 和一个 Exit。但是也可以有其他的控件，Dock 还可以用作简单的连接控件，如图 2.24 所示。

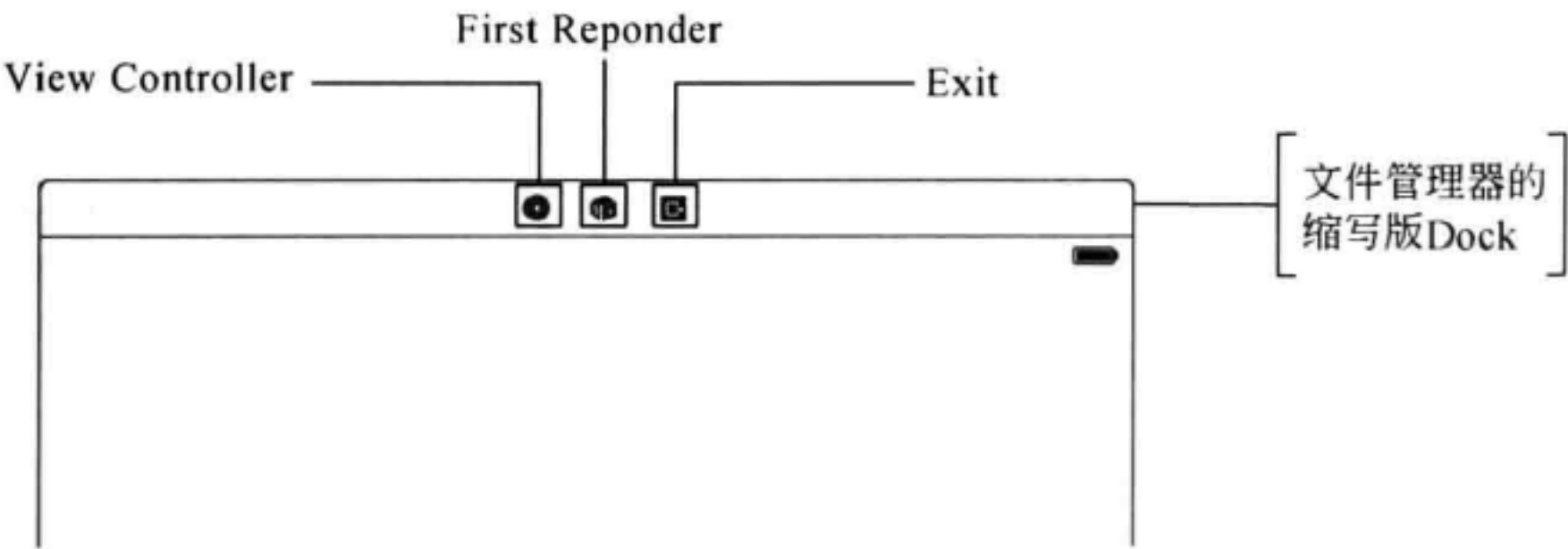


图 2.24 编辑界面


2.3.2 设计界面

在 2.1 节中提到了由于没有对编辑界面进行设计, 所以导致在运行的效果中没有任何内容。本小节将会在 iOS 模拟器上显示一个文本框。具体的操作步骤如下所述。

(1) 选择 Show the File inspector 选项, 将 Interface Builder Document 中的 Opens in 改为 Xcode 5.1, 如图 2.25 所示。



图 2.25 设计界面大小

注意: 由于 Xcode 6.0 的界面比较大, 为了方便截图, 将 Xcode 6.0 界面改为 Xcode 5.1 的界面。

(2) 单击工具窗口的库, 在其中找到 Show the Object Library, 即视图对象库窗口, 在里边找到 Text Field 文本框视图将其拖动到画布的界面中, 如图 2.26 所示。

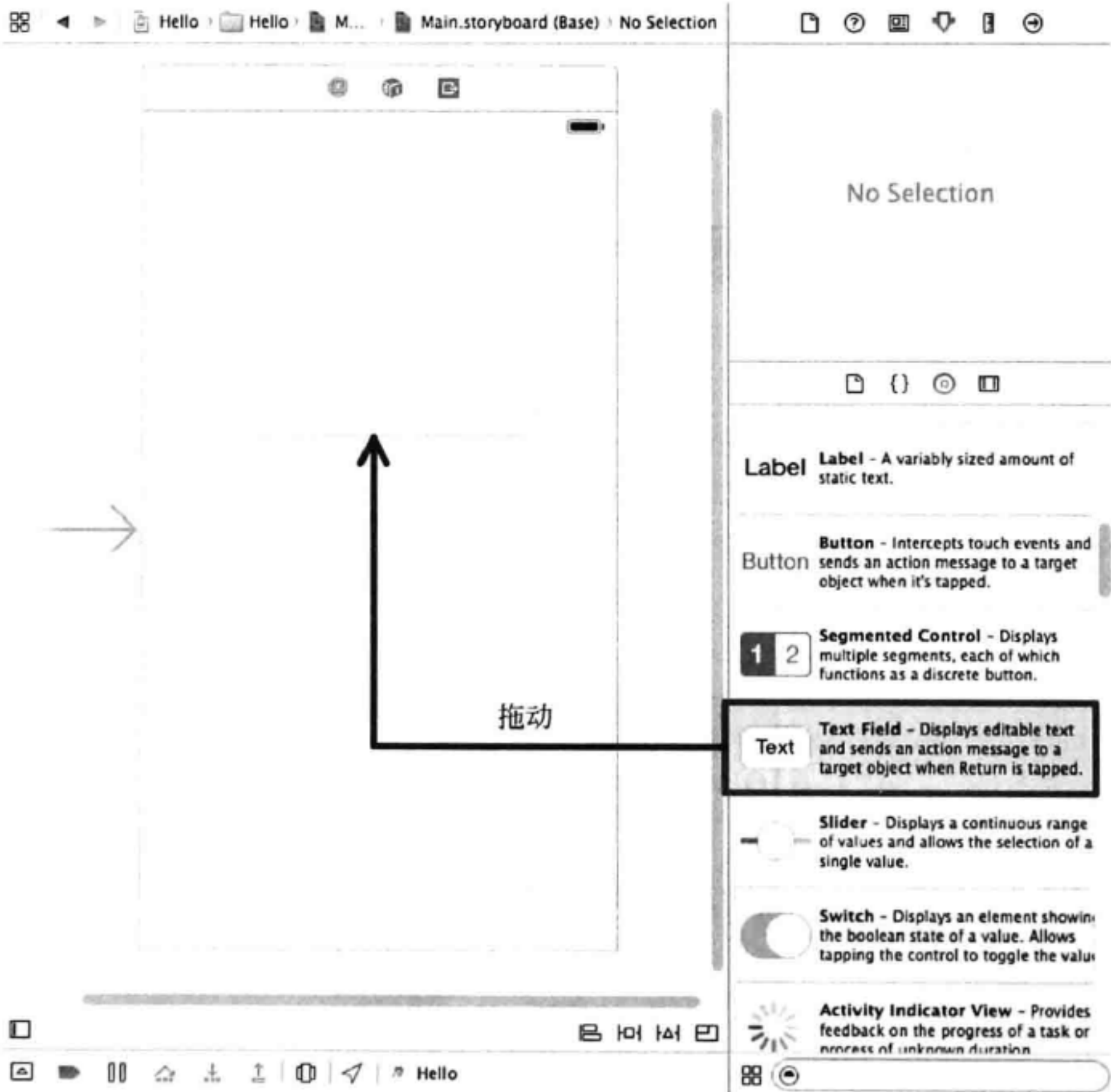


图 2.26 设计界面

此时运行程序, 可以看到如图 2.27 所示的效果。如果轻拍模拟器中的文本框, 还会出现一个虚拟键盘, 如图 2.28 所示。

轻按虚拟键盘中的键, 可以将用户按下的键显示在文本框中, 如图 2.29 所示。

文本框

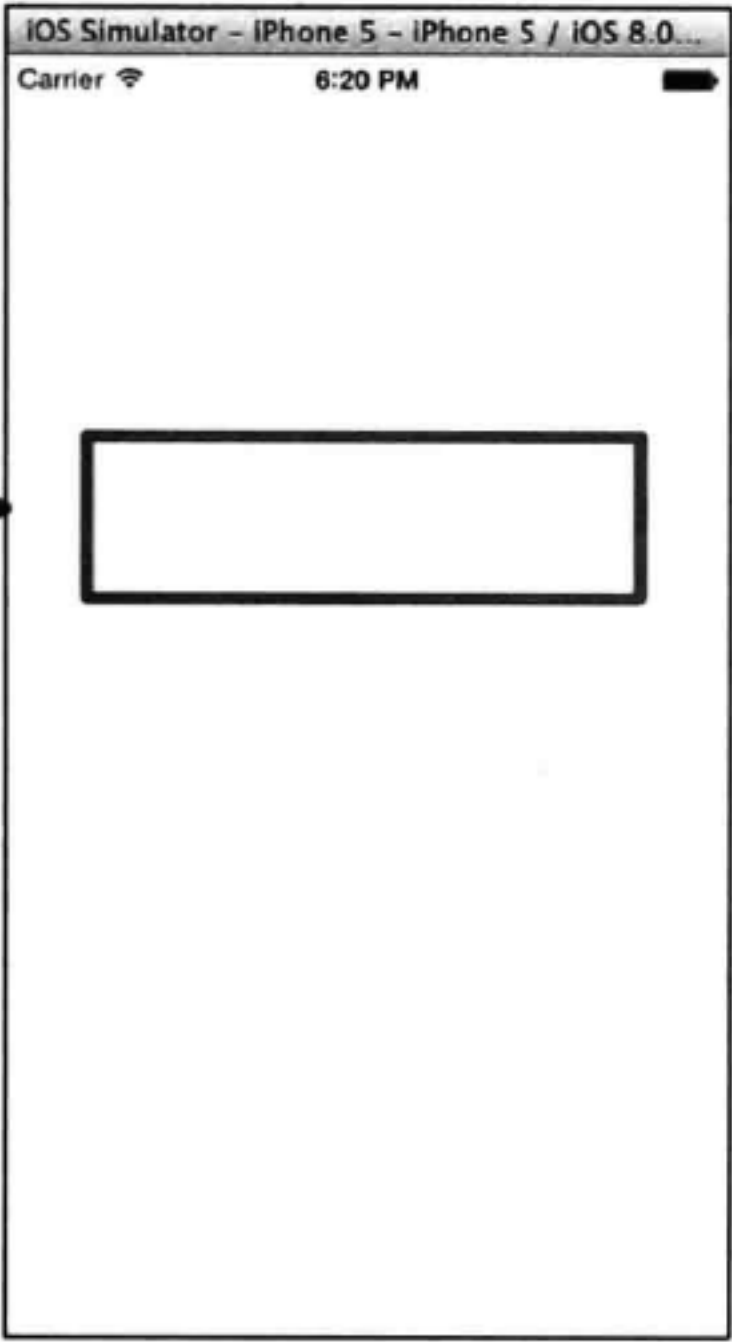


图 2.27 运行效果 1



图 2.28 运行效果 2



图 2.29 运行效果 3

⚠注意：文本框的显示风格、文本颜色及字体大小等都可以进行改变。开发者只需要打开 Show the Attributes inspector 选项，即属性查看器，就可以在此面板中进行文本框的属性设置。例如，将文本框的字体颜色设置为红色，将字体大小设置为 20，对齐方式设置为居中对齐，边框的风格设置为线框风格，如图 2.30 所示。



图 2.30 属性设置

此时运行程序，会看到如图 2.31 所示的效果。当用户轻拍文本框，并输入内容后，会看到如图 2.32 所示的效果。

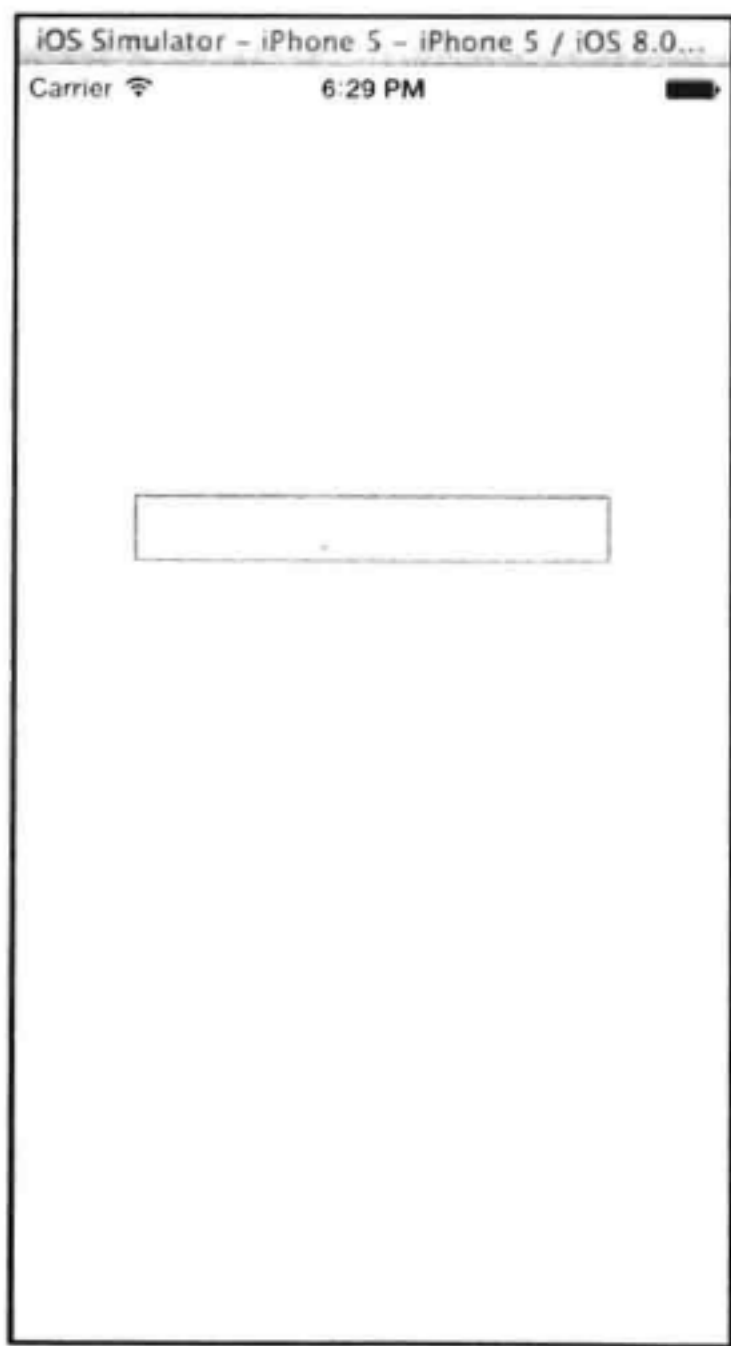


图 2.31 运行效果 1



图 2.32 运行效果 2

2.3.3 视图对象库的介绍

在设计界面一节中，我们提到了视图对象库。在视图对象库中存放了 iOS 开发中所需的所有视图，这些视图都是以扁平化风格设计的，如图 2.33 所示。



图 2.33 视图对象库

在视图对象库中存放的视图，可以根据功能的不同进行分类，如表 2-2 所示。

表 2-2 视图的分类

名 称	功 能
Controls（控件）	用于接收用户输入的信息
Data View（视图）	用于显示信息
Gesture Recognizers（手势识别器）	用于识别轻拍、轻扫、旋转和捏合
Objects&Controllers（控制器）	用于控制其他视图
Windows&Bars（其他）	用于显示其他各种视图

在视图对象库的最下边有两个图标，一个是用来进行显示视图排列方式的，另一个是用来搜索视图的，如图 2.34 所示。

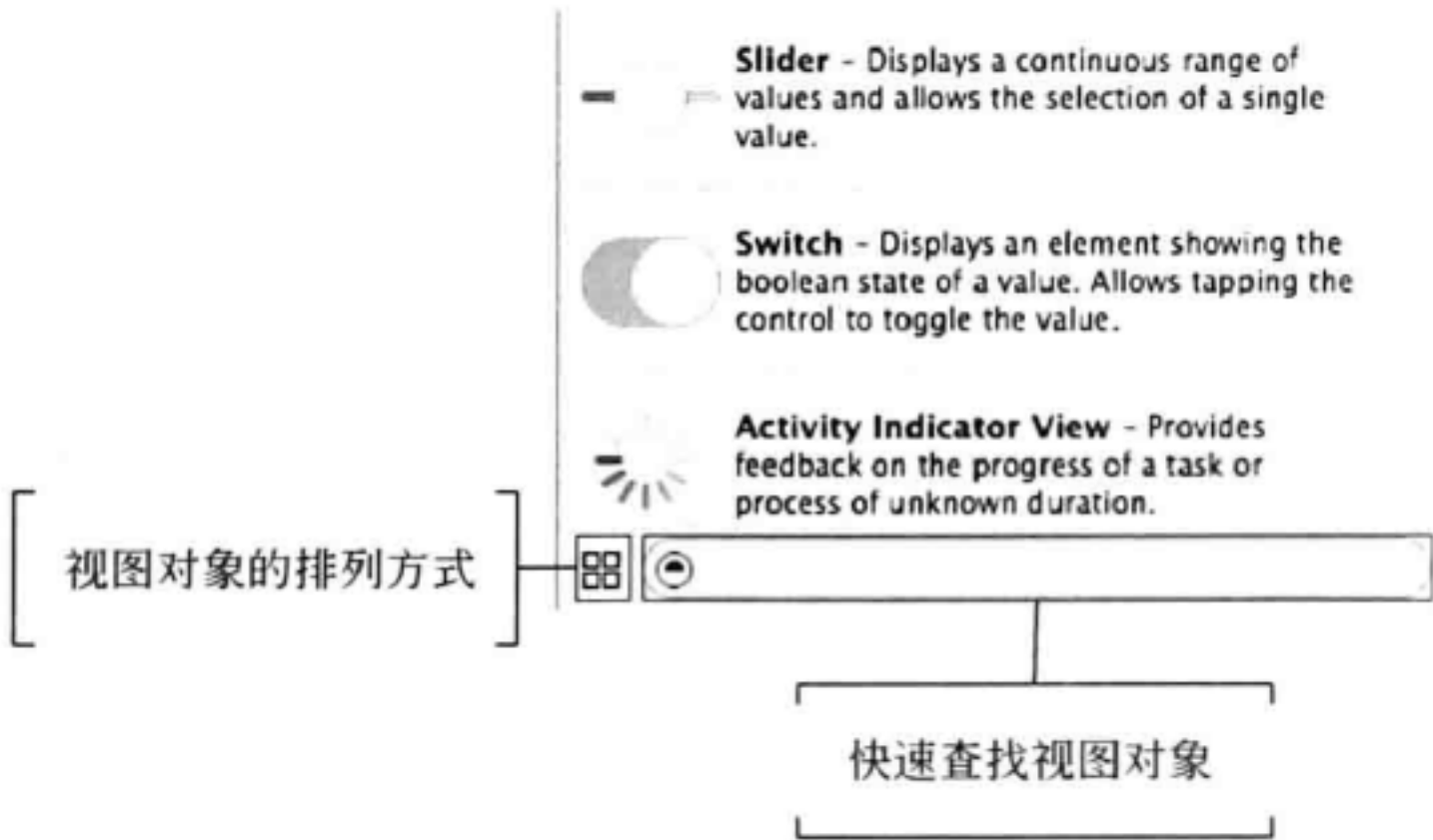


图 2.34 视图对象库

2.4 编写代码

代码，就是用来实现某一特定的功能而用计算机语言编写的命令序列的集合。现在就来通过代码实现在文本框视图对象中显示"Hello,World"字符串，操作步骤如下所述。

(1) 使用调整窗口中的第二个工具，即 Show the Assistant editor 工具，如图 2.35 所示，将 Xcode 的界面调整为如图 2.36 所示的效果。



图 2.35 调整窗口

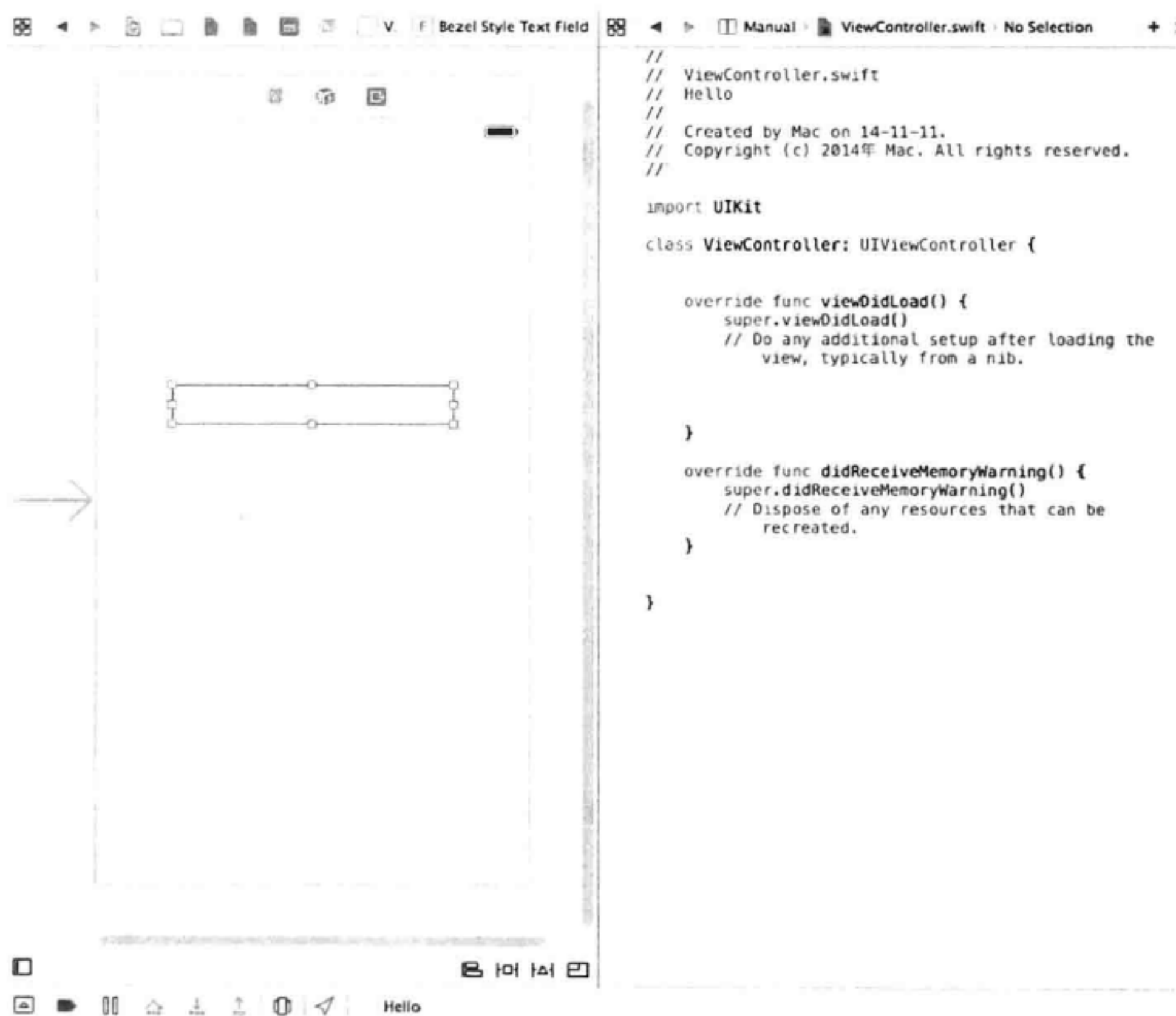


图 2.36 操作步骤 1

(2) 按住 Ctrl 键拖动界面中的文本框对象，这时会出现一个蓝色的线条，将这个蓝色的线条拖动到 ViewController.swift 文件中，如图 2.37 所示。

(3) 松开鼠标后，会弹出一个对话框，如图 2.38 所示。

(4) 在弹出的对话框中，找到 Name 这一项，在其文本框中输入名称，如图 2.39 所示。

(5) 单击 Connect 按钮，关闭对话框，这时在 ViewController.swift 文件中自动生成一行代码，如图 2.40 所示。

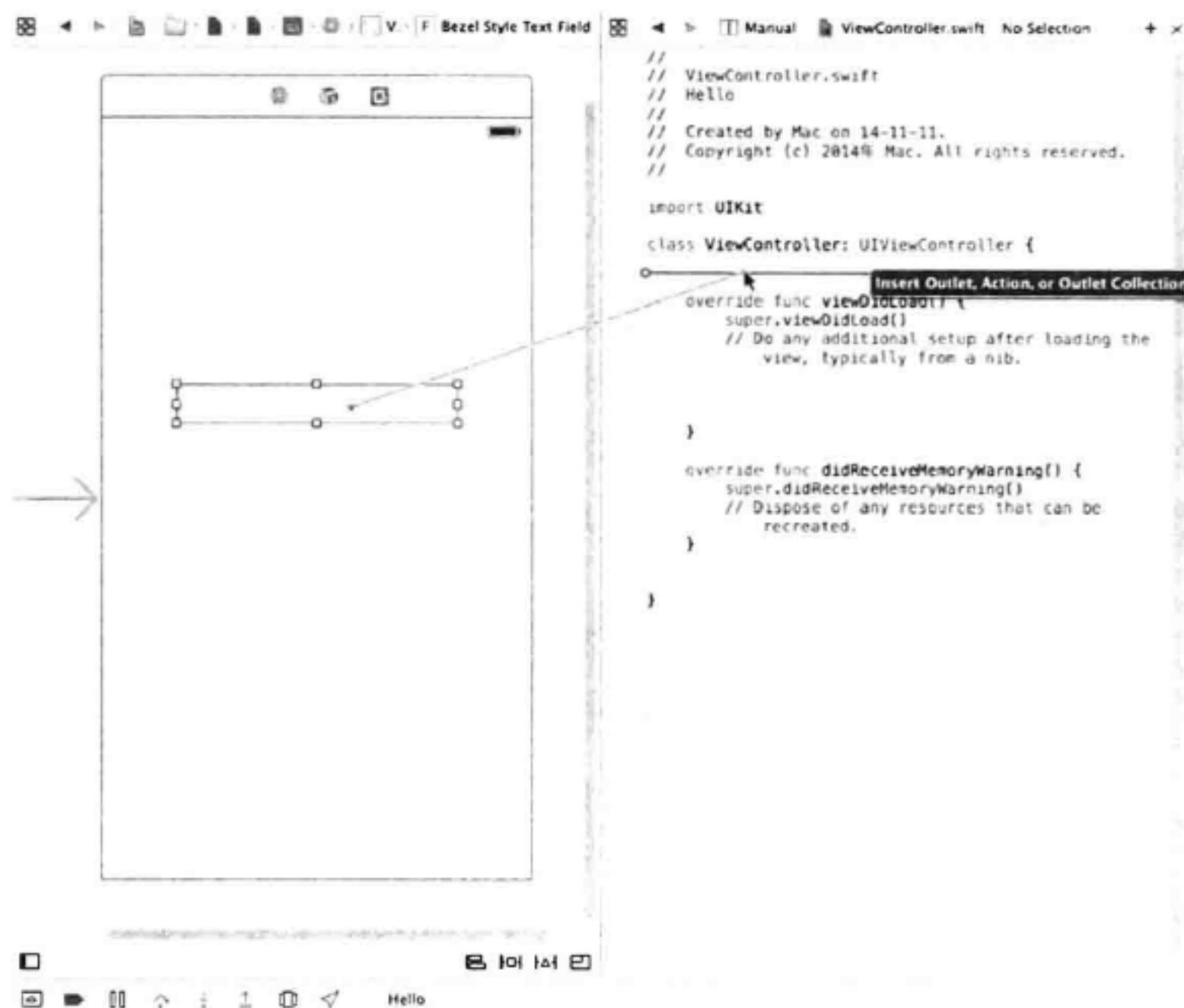


图 2.37 操作步骤 2



图 2.38 操作步骤 3



图 2.39 操作步骤 4

⚠注意：生成的代码叫做插座变量，插座变量其实就是为关联的对象起了一个别名，开发者就可以对此插座变量进行操作，从而对关联的对象进行操作。

(6) 打开 ViewController.swift 文件，编写代码，实现在文本框中显示字符串 "Hello,World" 的功能，代码如下：

```
import UIKit
class ViewController: UIViewController {
    @IBOutlet weak var tf: UITextField!
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
        tf.text="Hello,World" //显示的文本内容
    }
    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

此时运行程序，会看到如图 2.41 所示的效果。

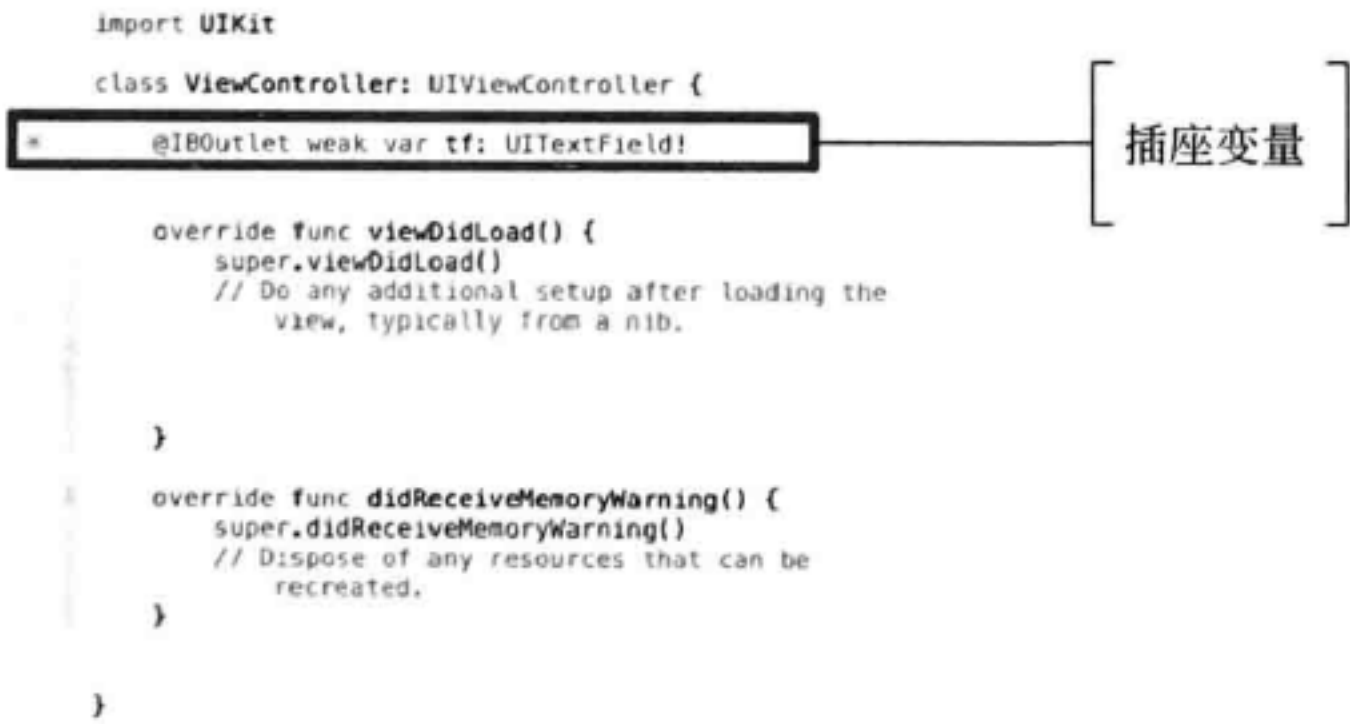


图 2.40 操作步骤 5



图 2.41 运行效果

2.5 调 试

调试又被称为排错，是发现和减少程序错误的一个过程。在 Xcode 中进行调试，需要实现以下几个步骤。

1. 添加断点

在进行程序调试之前，首先需要为程序添加断点，断点是指可以暂停调试器中程序的运行，并可以让开发者查看程序的地方。将光标移动到要添加断点的地方，通过快捷键 `Command+\` 或者选择菜单栏中的 `Debug|Breakpoints|Add Breakpoint at Current Line` 命令进行断点的添加，之后会在添加断点代码的最左边看到一个蓝色箭头，这就是一个新断点，如图 2.42 所示。



图 2.42 添加断点

2. 运行程序

单击运行按钮后，程序就会运行，这时运行的程序会停留在断点所在的位置，并且此代码行会出现绿色的箭头，表示现在程序运行到的位置，如图 2.43 所示。不仅如此，iOS 模拟器也会显示，但是没有内容。



图 2.43 执行断点

3. 断点导航

在程序停留下来后，程序调试信息窗口就会出现，里面显示了一些调试信息。在程序调试信息窗口顶端，会出现断点导航，如图 2.44 所示。

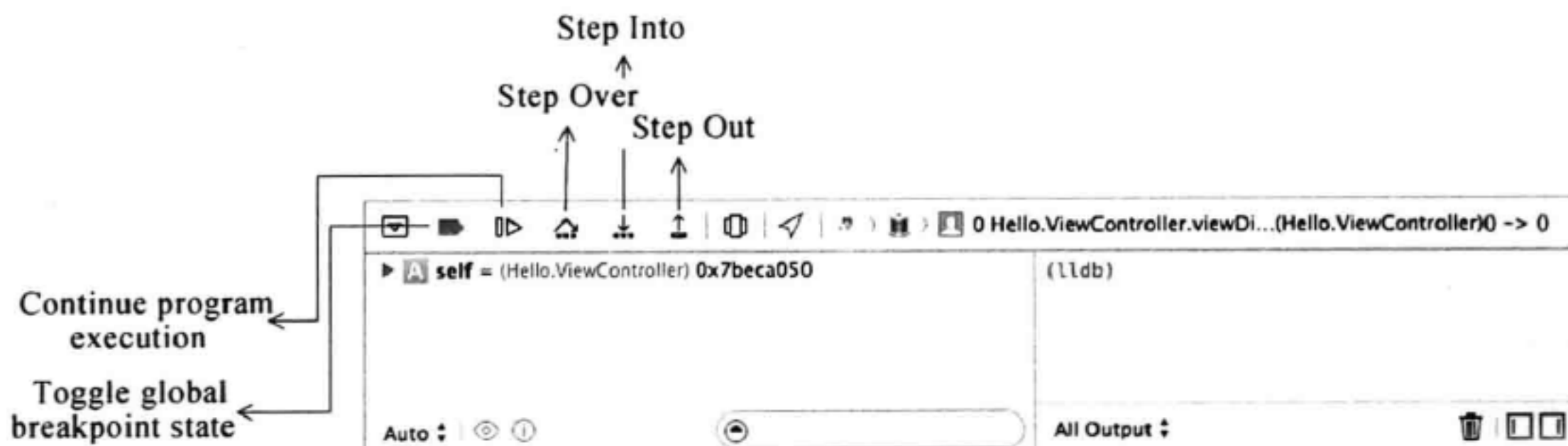


图 2.44 断点导航

- ❑ **Toggle global breakpoint state 按钮：**选择要查看的线程。
- ❑ **Continue program execution 按钮：**继续执行当前的代码，如果有下一个断点，就停止在下一个断点上。
- ❑ **Step Over 按钮：**执行下一个代码。如果当前行是方法调用，则不会进入方法内部。
- ❑ **Step Into 按钮：**进入方法内容。
- ❑ **Step Out 按钮：**跳过当前方法，即执行到当前方法的末尾。

这时，单击断点导航中的 Continue program execution 按钮，继续执行当前的代码。如果这时程序出现错误就不会跳到下一断点处；如果程序没有问题就会继续向下执行。现在只有一个断点，单击此按钮，程序会输出最后的结果。

4. 删除或废弃断点

如果程序没有问题，那么就要将程序中设置的断点进行删除或者废弃。删除断点常用的方法有 3 种：

- ❑ 右击设置的断点，在弹出的快捷菜单中选择 Delete Breakpoint 命令。
- ❑ 选中设置断点的行，在 Xcode 的菜单栏中选择 Debug|Breakpoints|Remove Breakpoint at Current Line 命令。
- ❑ 选择断点，将其拖动到别的地方，这时，此断点就被删除了。

废弃断点，就是单击断点，这时，断点就由深蓝色变为了浅蓝色。浅蓝色的断点就说明该断点已被废弃，如图 2.45 所示。

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var tf: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
        tf.text="Hello,World"
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

图 2.45 废弃断点

2.6 真机测试

在 2.2.1 节中提供的模拟器只可以模拟 iPhone 或者是 iPad 的部分功能。对于一些无法实现的功能，还需要使用真机，如打电话、发送短信、定位功能以及照相等。本节将讲解如何使用真机进行程序的测试。

2.6.1 申请和下载证书

申请和下载证书的具体步骤如下所述。

1. 创建App ID

在申请和下载证书之前，首先要创建一个 App ID。App ID 是一系列字符，用于唯一标识 iOS 设备中的应用程序。创建 App ID 的具体步骤如下所述。

(1) 在 Safari 的搜索栏中输入网址 (<https://developer.apple.com/devcenter/ios/index.action>)，然后按回车键，进入 iOS Dev Center-App Developer 网页，如图 2.46 所示。



图 2.46 操作步骤 1

(2) 单击 Log in 按钮，进入 Sign in with your Apple ID-Apple Developer 网页，在此网页中需要开发者输入 App ID 以及密码，然后单击 Sign In 按钮，此时会再次进入 iOS Dev Center-App Developer 网页，如图 2.47 所示。

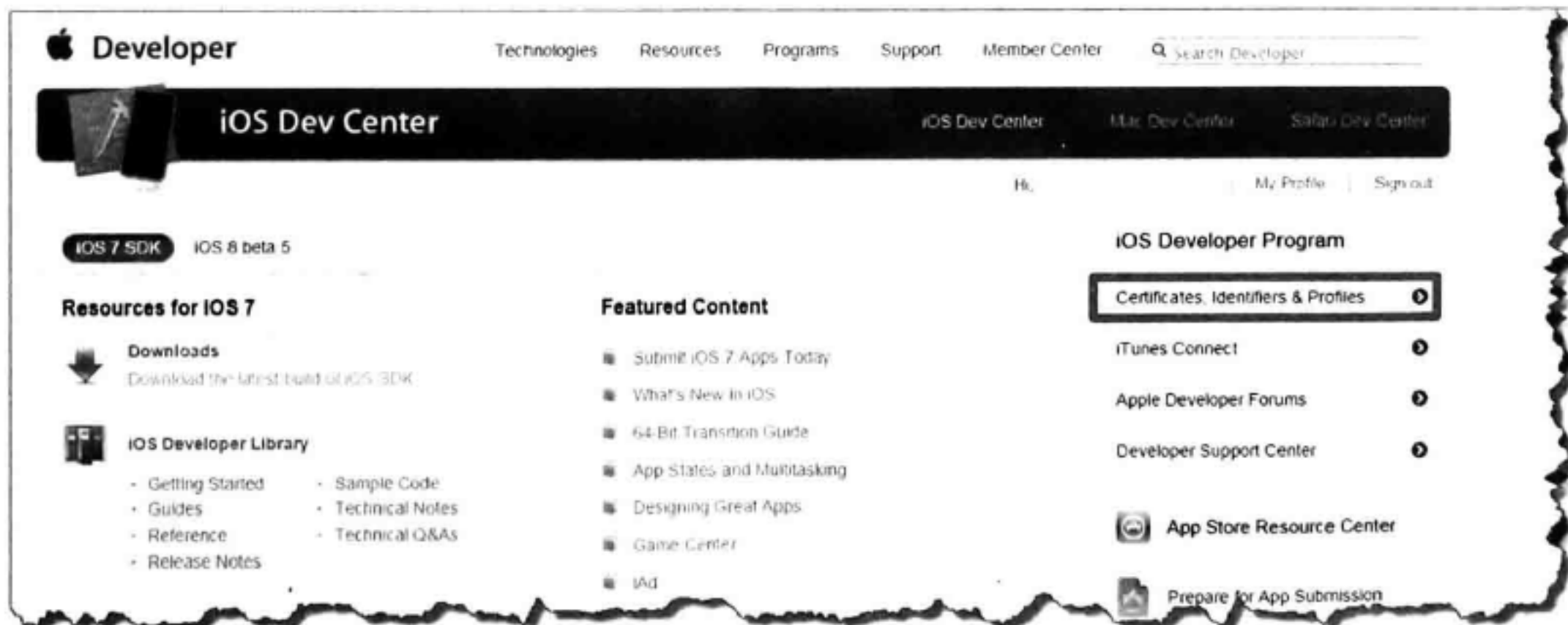


图 2.47 操作步骤 2

🔔注意：图 2.47 所看到的网页只有申请付费开发者账号后，才可以看到。

(3) 选择 Certificates, Identifiers & Profiles 选项，进入 Certificates, Identifiers & Profiles-App Developer 网页，如图 2.48 所示。



图 2.48 操作步骤 3

(4) 选择 Identifiers 选项，进入 iOS App IDs-Apple Developer 网页，在此网页中，选择蓝色的 Register your App ID 字符串，进入 Register-iOS App IDs-Apple Developer 网页，在此网页中填入一些相关的内容。这些内容分为 4 部分，分别为 App ID Description、App ID Prefix、App ID Suffix 和 App Services。在填写 App ID Suffix 这部分内容时需要特别注意，如图 2.49 所示。

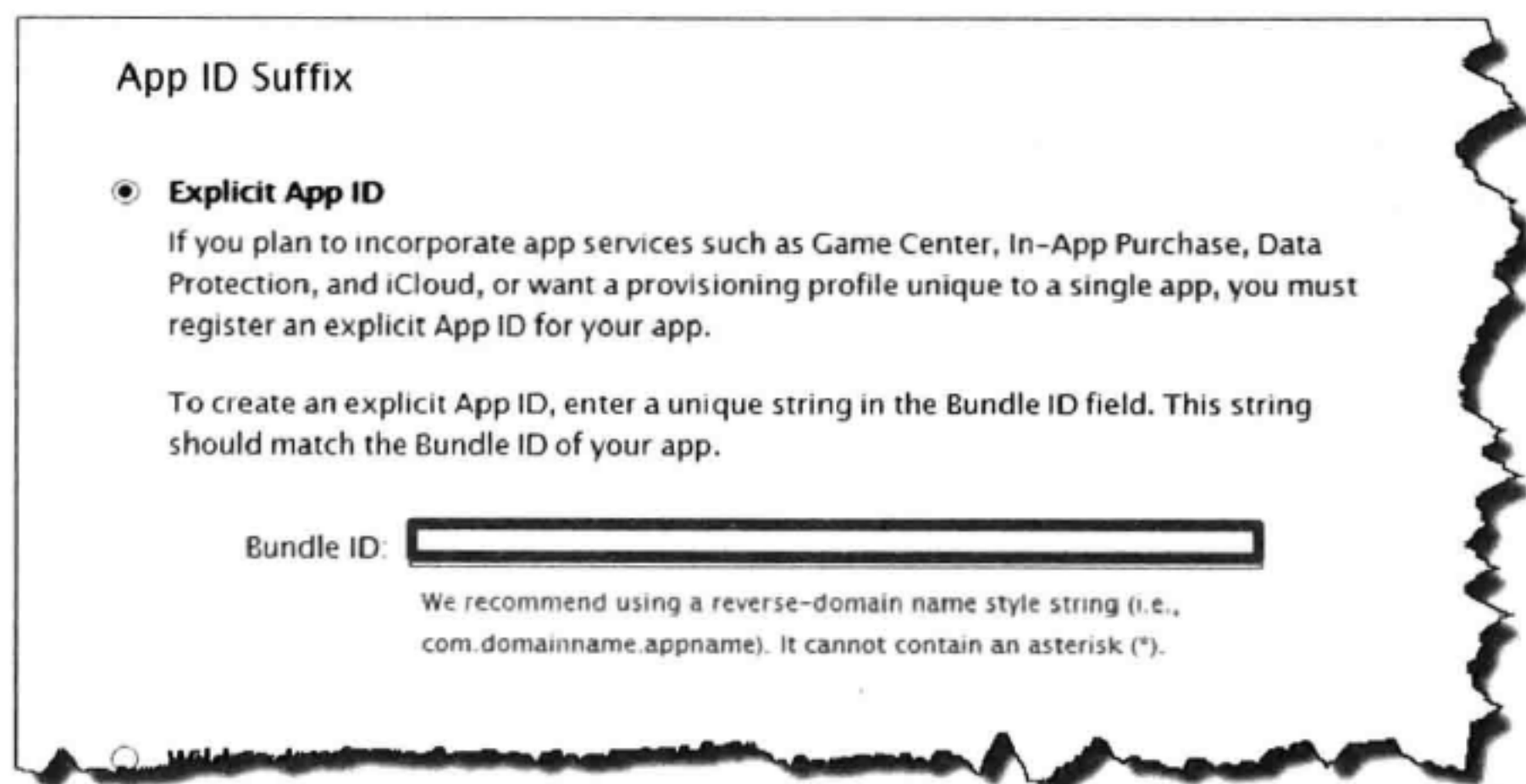


图 2.49 操作步骤 4

(5) 单击 Continue 按钮，进入 Add-iOS App IDs-Apple Developer 网页，然后单击 Submit 按钮，之后再单击 Done 按钮。这样，一个 App ID 就创建好了。

2. 获取设备的 UDID

将设备连接到 Mac(或者 Mac 虚拟机)上，启动 Xcode。在菜单栏中选择 Window|Devices 命令，弹出 Devices 对话框，如图 2.50 所示。在对话框中显示的就是开发者的设备信息，其中 Identifier 就是 UDID。

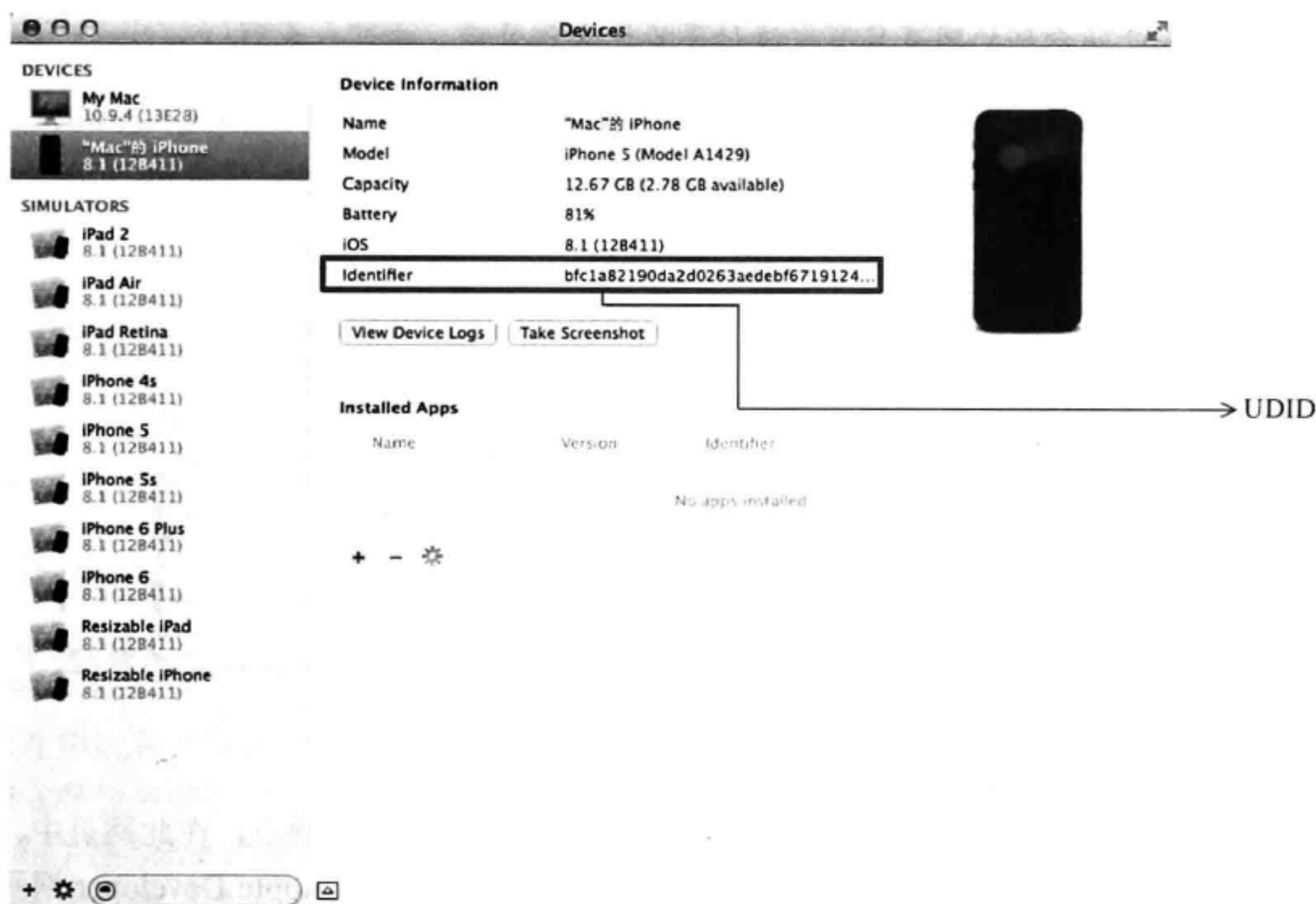


图 2.50 操作步骤

3. 注册设备

如果开发者的设备是连接在 Mac (或者 Mac 虚拟机) 上的, 回到 Certificates, Identifiers & Profiles-App Developer 网页, 选择 Devices 选项, 或者是如果开发者还处于创建 App ID 的网页, 可以选择此网页右侧的 Devices 下的 All 选项, 都会进入 iOS Devices-Apple Developer 网页, 并会看到连接在 Mac (或者 Mac 虚拟机) 上的设备已经被注册好了, 如图 2.51 所示。



图 2.51 操作步骤

注意: 如果开发者还需要注册其他的设备, 可以单击添加设备的按钮, 对设备进行添加, 如图 2.52 所示。

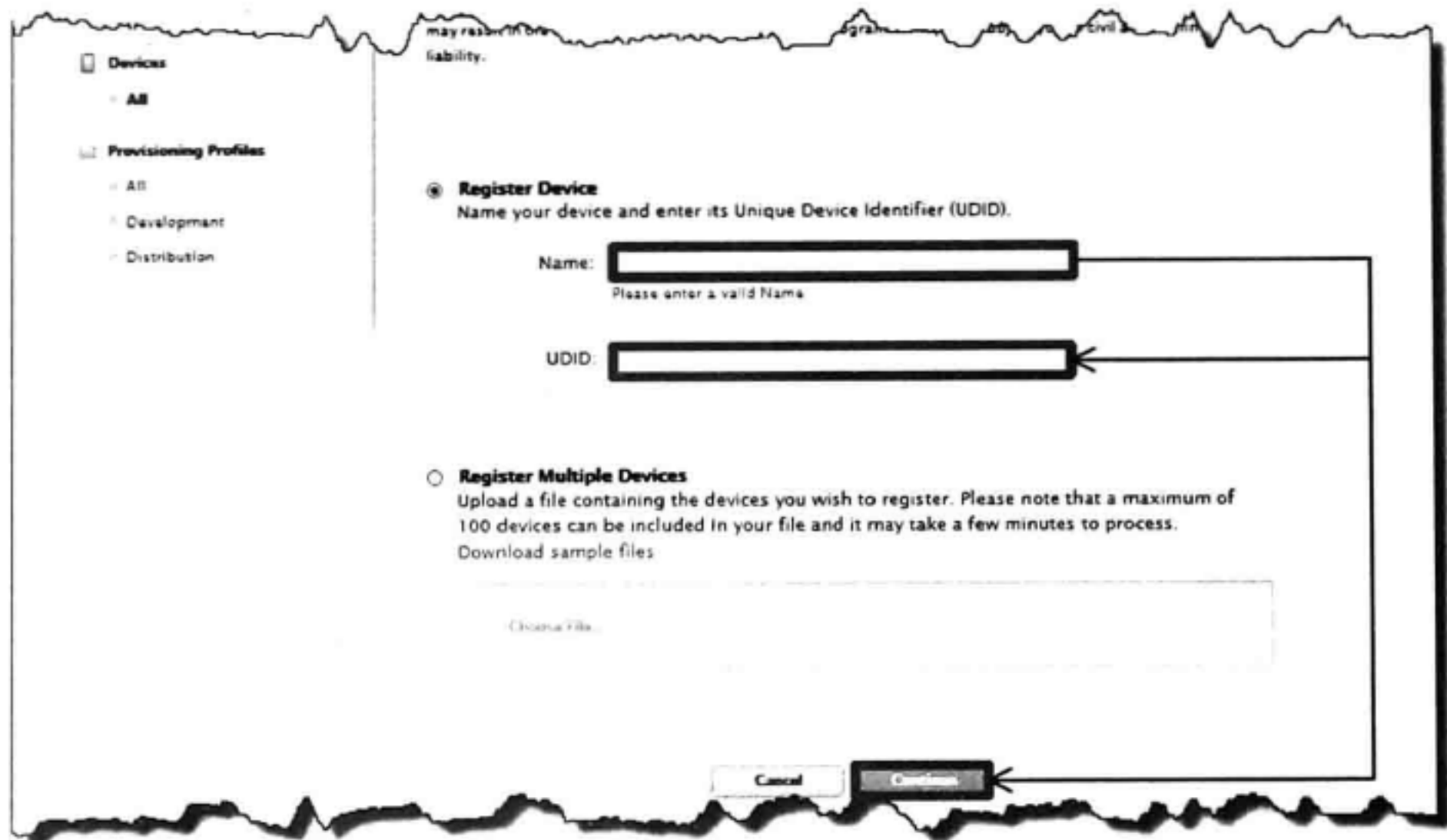


图 2.52 注册新的设备

在此图中，开发者只需要在 Name 文本框中输入设备的名称，在 UDID 文本框中输入设备的标识符就可以了，然后单击 Continue 按钮，进入对设备的检测和登记网页，再单击 Register 按钮，进入登记设备成功的网页，最后单击 Done 按钮，一个新的设备就注册成功了。

4. 生成证书签名申请

为了从 Apple 公司申请开发证书，需要生成一个证书签名申请。生成一个证书签名申请的具体步骤如下。

(1) 选择菜单栏中的“前往”|“实用工具”命令，到“实用工具”文件夹中，如图 2.53 所示。



图 2.53 操作步骤 1

(2) 找到“钥匙串访问”应用程序，双击图标，将其打开，选择菜单栏上的“钥匙串访问”命令，如图 2.54 所示。



图 2.54 操作步骤 2

(3) 选择“证书助理”|“从证书颁发机构请求证书...”命令，弹出“证书助理”对话框，如图 2.55 所示。

(4) 输入用户电子邮件地址、选择存储到磁盘复选框，然后单击“继续”按钮，弹出存储位置对话框如图 2.56 所示。

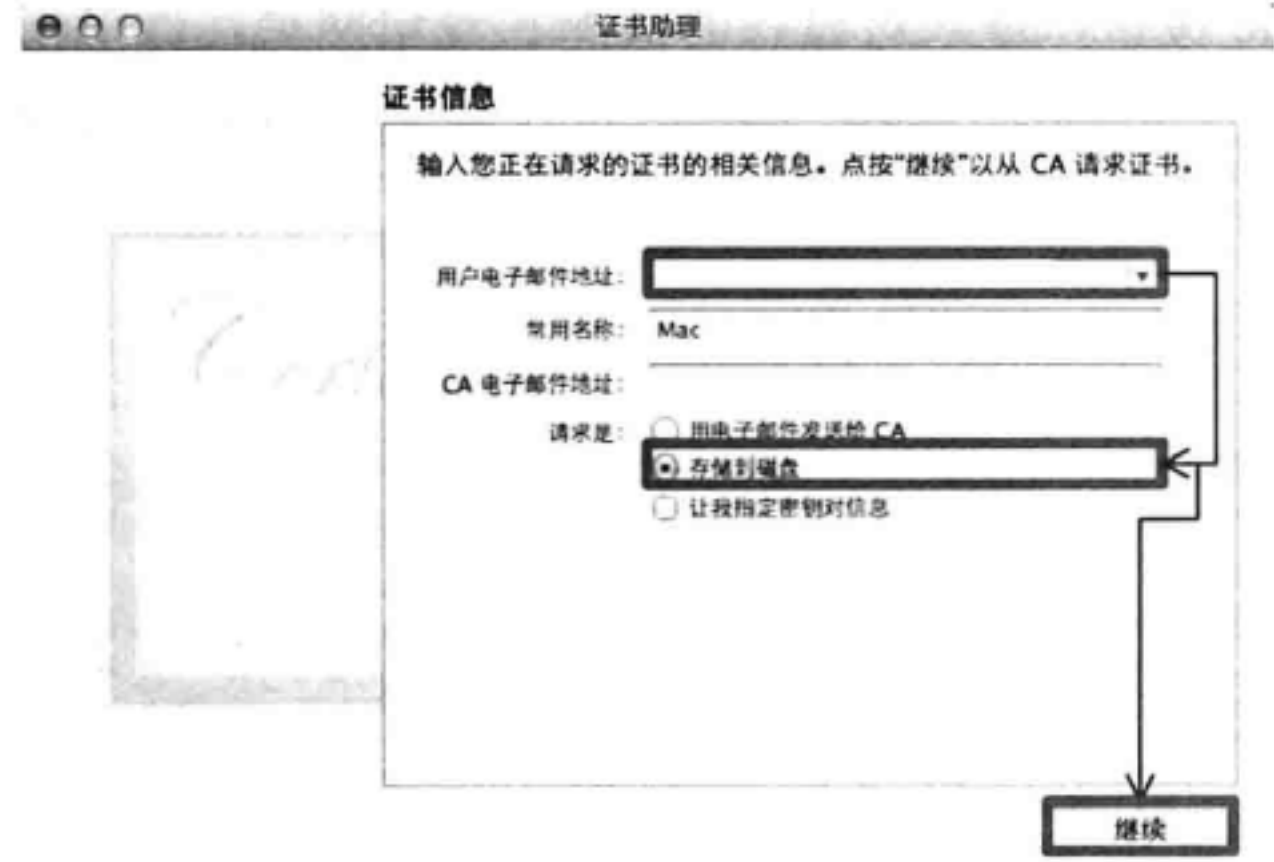


图 2.55 操作步骤 3



图 2.56 操作步骤 4

注意：在存储位置对话框中，“存储为”以及“位置”下拉列表都有默认选项。

(5) 设置“位置”为桌面，然后单击“存储”按钮，就在桌面生成了一个证书签名申请，并回到“证书助理”对话框，告诉开发者证书请求已经在磁盘上创建好了，此时单击“完成”按钮即可。

5. 生成证书

以上这些准备工作都做好后，便可以生成证书了，它包括了证书的申请和下载。具体的操作步骤如下。

(1) 如果开发者还处于注册设备的网页，可以选择此网页右侧的 Certificates 的 Development 选项，进入 iOS Certificates (Development)-Apple Developer 网页，如图 2.57 所示。



图 2.57 操作步骤 1

(2) 选择 iOS App Development 复选框，单击 Continue 按钮，进入 Request 选项卡的网页中。在此网页中，单击 Continue 按钮，进入 Generate 选项卡的网页中，如图 2.58 所示。

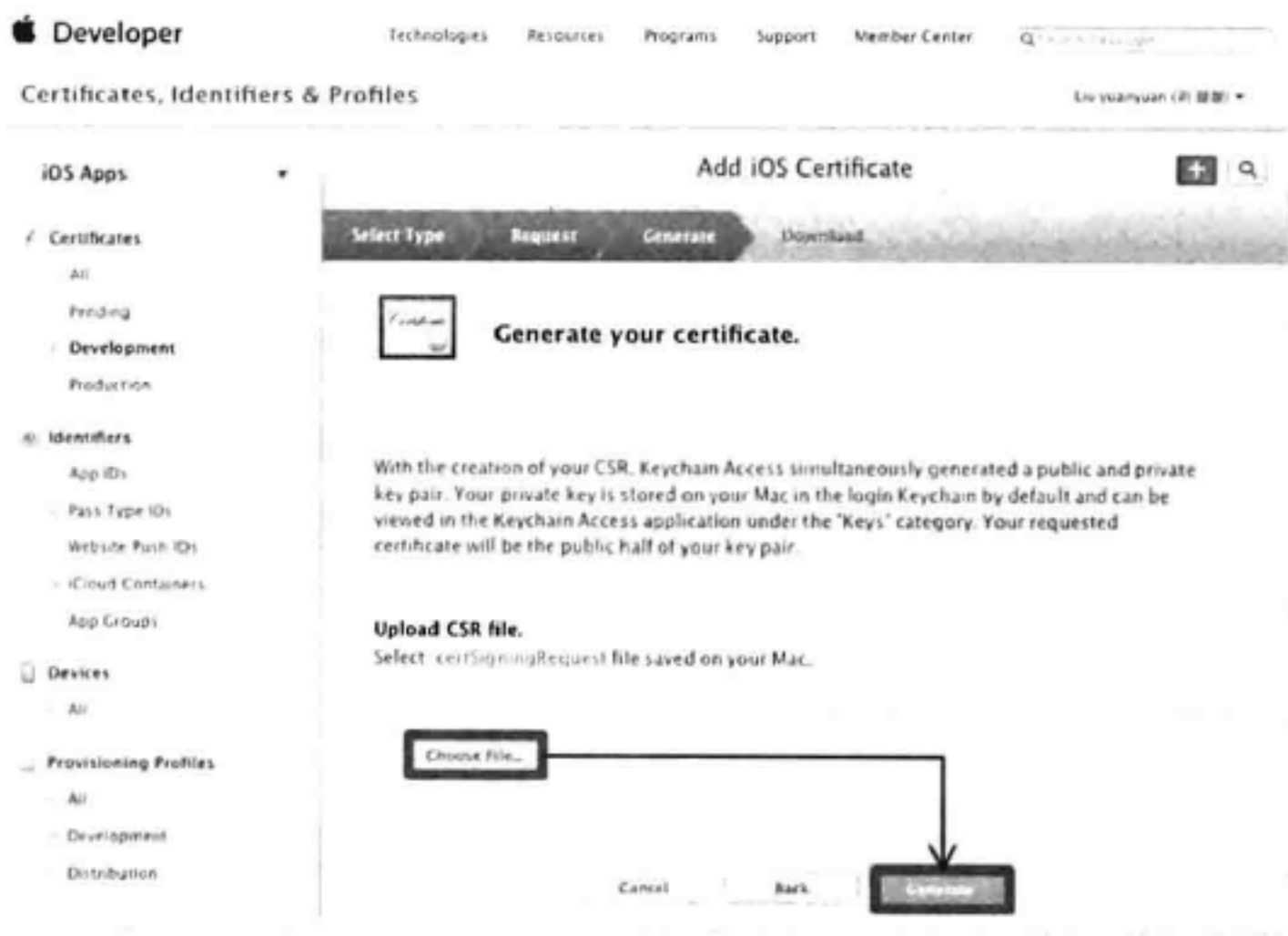


图 2.58 操作步骤 2

(3) 选择 Choose File...按钮后，弹出选择文件对话框，如图 2.59 所示。

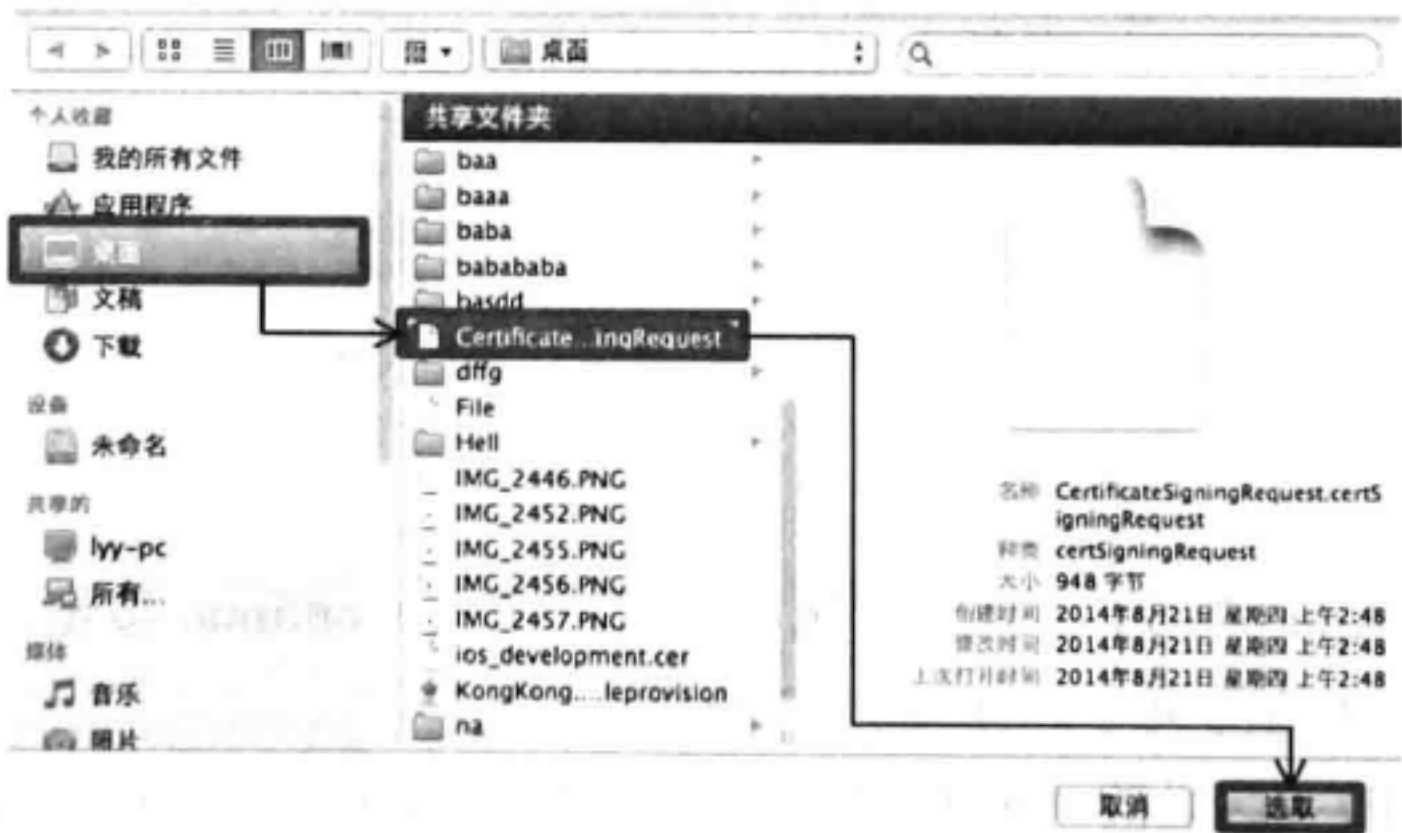


图 2.59 操作步骤 3

(4) 选择在桌面的 CertificateSigningRequest.certSigningRequest 文件, 此文件就是生成的证书签名申请, 单击“选取”按钮, 再单击 Generate 按钮, 进入到 Download 选项卡的网页中, 如图 2.60 所示。

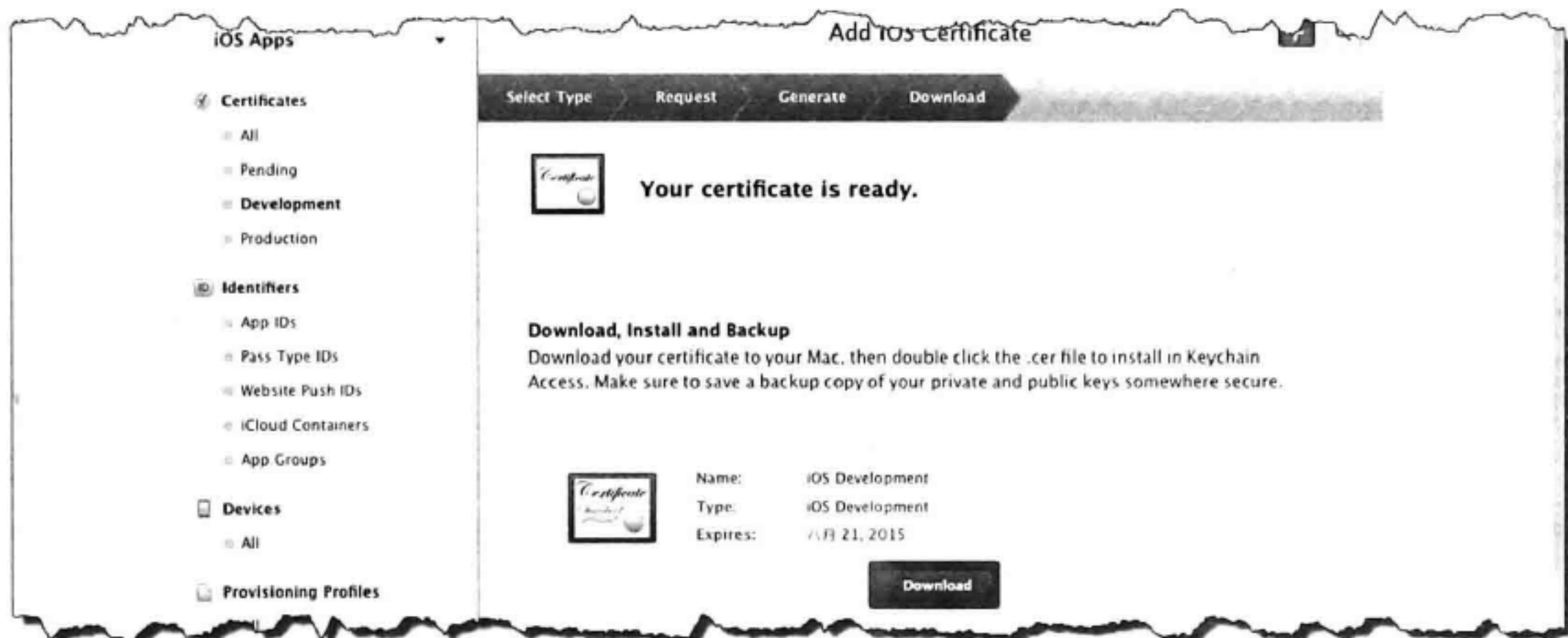


图 2.60 操作步骤 4

(5) 单击 Download 按钮, 对生成的证书进行下载。下载后的证书名为 ios_development.cer。

(6) 如果开发者还处于下载证书的网页, 可以选择此网页右侧的 Provisioning Profiles 的 Development 选项, 进入 iOS Provisioning Profiles (Development)-Apple Developer 网页。在此网页中, 选择蓝色的 manually generate profiles 字符串, 进入 Add-iOS Provisioning Profile-Apple Developer 网页, 如图 2.61 所示。

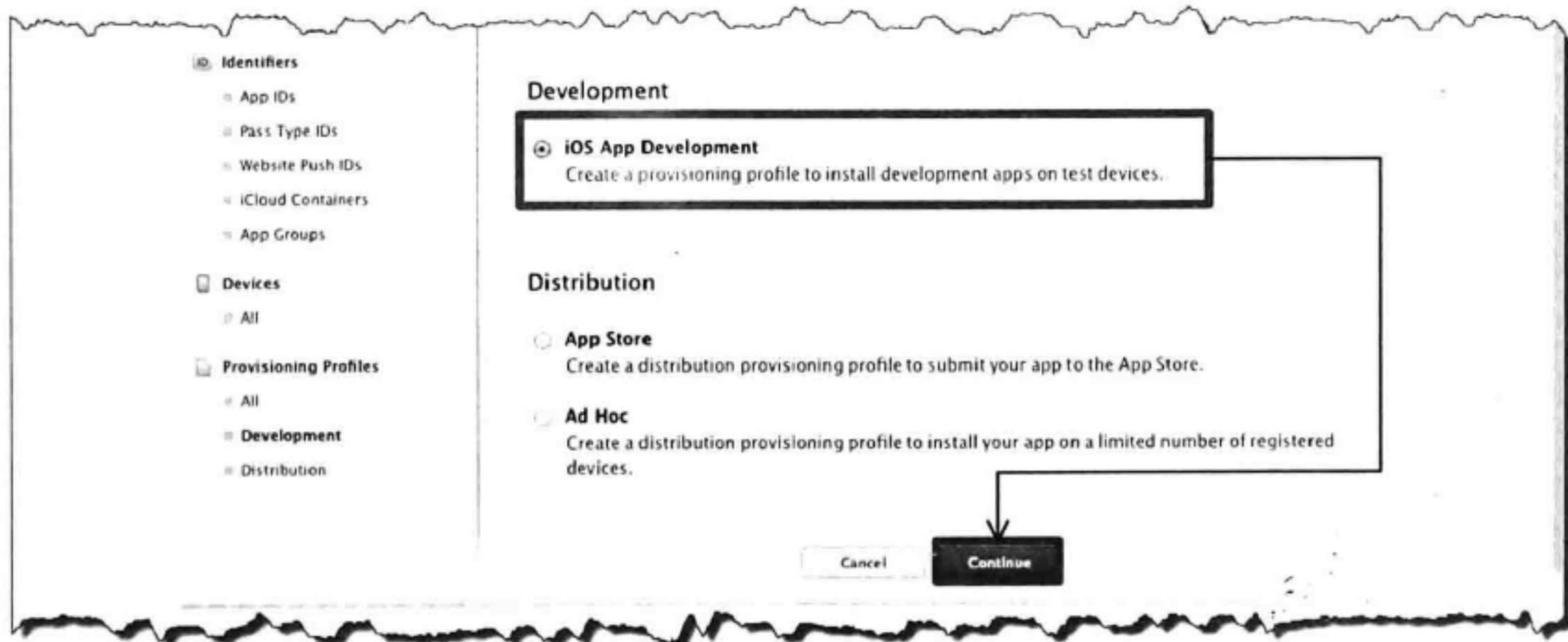


图 2.61 操作步骤 5

(7) 选择 iOS App Development 复选框, 然后单击 Continue 按钮, 进入到 Configure 选项卡的选择 App ID 的网页中, 如图 2.62 所示。

(8) 选择 App ID (这里的 App ID 是之前创建的 App ID), 然后单击 Continue 按钮, 进入 Configure 选项卡的选择证书网页中, 如图 2.63 所示。



图 2.62 操作步骤 6



图 2.63 操作步骤 7

(9)选择 Select All 复选框或者选择某一个证书, 然后单击 Continue 按钮, 进入 Configure 选项卡的选择设备网页中, 如图 2.64 所示。



图 2.64 操作步骤 8

(10) 选择 Select All 复选框或者选择某一个设备, 然后单击 Continue 按钮, 进入 Generate 选项卡的网页中, 如图 2.65 所示。



图 2.65 操作步骤 9

(11) 输入配置的文件名, 然后单击 **Generate** 按钮, 进入 **Download** 选项卡的网页中, 如图 2.66 所示。



图 2.66 操作步骤 10

(12) 单击 **Download** 按钮, 对 **Provisioning Profiles** 进行下载, 下载后的文件为 **KongKong.mobileprovision**。

(13) 双击下载的 **ios_development.cer** 证书, 弹出“添加证书”对话框, 如图 2.67 所示。



图 2.67 操作步骤 11

(14) 单击“添加”按钮, 将下载的 **ios_development.cer** 证书添加到钥匙串中。

(15) 双击下载的 **KongKong.mobileprovision** 文件, 将此文件添加到 **Organizer** 的

Provisioning Profiles 中。

2.6.2 实现真机测试

在进行真机测试之前，首先需要确保设备已经连在了 Mac（或者 Mac 虚拟机）上，在 2.6.1 节开始，设备就一直连接在 Mac（或者 Mac 虚拟机）上，并且此设备就是注册过的。打开创建的项目，在运行按钮一栏中，将程序运行的设备设置为真机的名称。它会自动加载到“选择程序运行的设备”这一项中，如图 2.68 所示。单击“运行”按钮，就可以看到应用程序在真机上运行了。



图 2.68 设置设备

2.7 使用帮助文档

在编写代码的时候，可能会遇到很多的方法。如果开发者对这些方法的功能，以及参数不是很了解，就可以使用帮助文档。那么帮助文档该如何打开以及如何查找相关的内容呢。以下将解决这些问题。

1. 打开帮助文档

要使用帮助文档，必须要对其进行打开。选择 Help|Documentation and API Reference 命令，就可以打开了，如图 2.69 所示。



图 2.69 帮助文档

2. 查找

如果想要查找一个方法，可以在搜索栏中输入这个方法，如图 2.70 所示。按回车键后，便可以找到相应的内容。

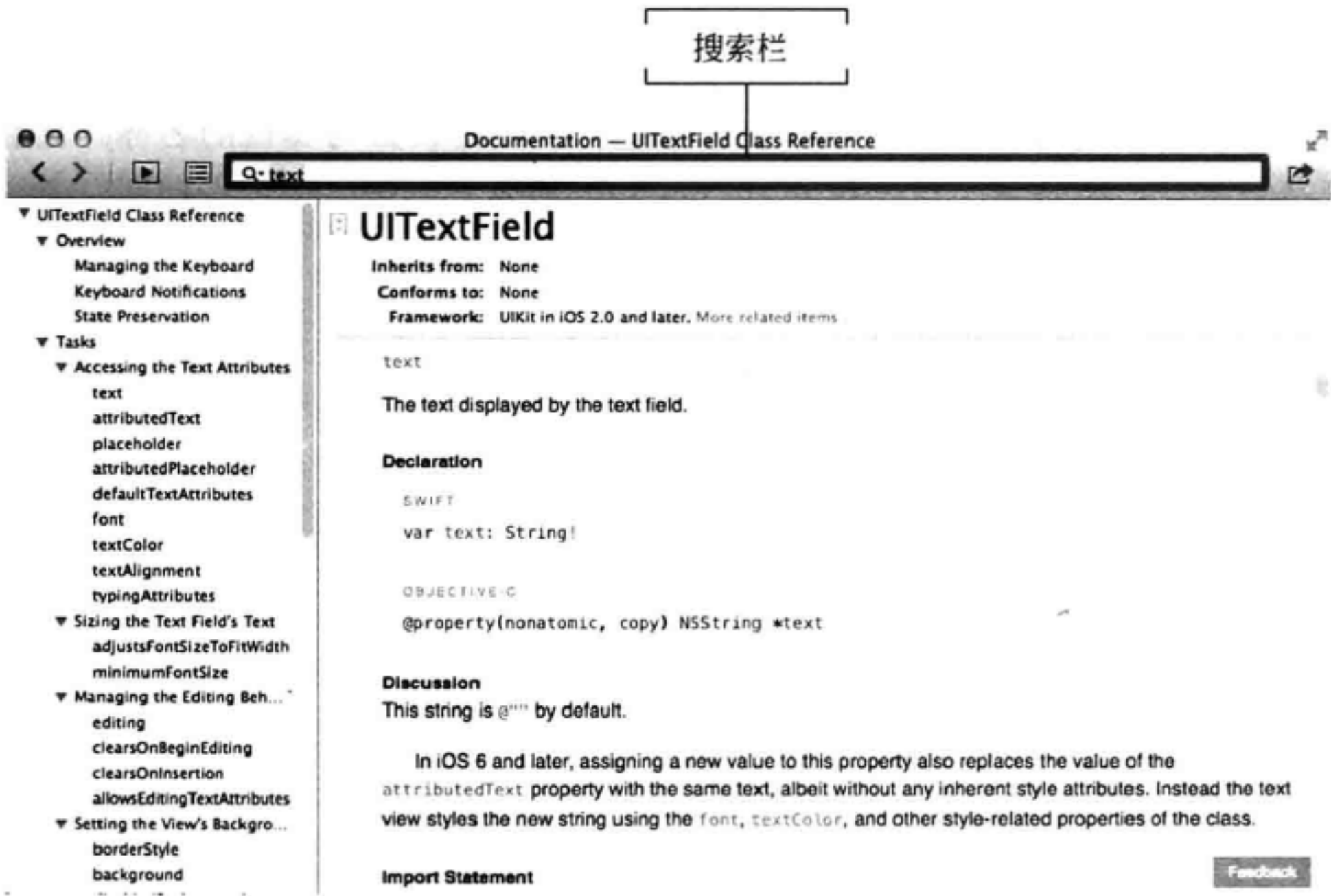


图 2.70 搜索内容

第 3 章 Swift 基础语法

Swift 语言是 2014 年 6 月 4 日苹果公司在 WWDC 开发者大会上推出的新的编程语言。此语言具备了 Objective-C 语言和 C 语言的特征。除此之外，此语言还具有自身的 6 个特点：安全、流行、强大、交互性、高效和兼容。本章将首先讲解 Swift 的基础语法，如常量、变量、数据类型、运算符、程序控制结构和函数等内容。

3.1 常量和变量


常量和变量在每一个编程语言中都是会出现的。本节将讲解如何在 Swift 中使用常量和变量。

3.1.1 常量

在程序运行期间，不可以改变的量被称为常量。常量的值不需要在编译时指定，但至少赋值一次。常量在使用之前必须对其进行声明和定义。声明用来说明该标识符被作为一个常量来使用，定义是指定该常量所指代的数据类型。由于 Swift 支持类型推断，所以可以省略对常量指定数据类型。因为常量的声明和定义是同时进行的，所以将常量的声明和定义合并称为常量的定义。此时定义常量的语法形式如下：

```
let 常量名=值
```

其中，let 是定义常量的关键字；常量名是常量的名称；值是常量被赋的值。

 **注意：**常量名必须符合标识符命名规范。所谓标识符是用户编程时使用的名字。在计算机语言中，对于变量、常量和函数（对于函数会再后面的节中讲解）都有自己的名字。我们称这些名字为标识符。在 Swift 中，标识符分为两类：一类是用户标识符，另一类是关键字。首先，来看用户标识符。所谓用户标识符就是用户根据需要定义的标识符。一般来给变量和常量等进行命名。一个用户标识符命名是有一定的规则，如图 3.1 所示。

在图 3.1 中我们提到了不可以将 Swift 关键字作为标识符。其中，关键字是对编译器具特殊意义的预定义保留标识符。在 Swift 中，保留关键字是因为使用它们可以使代码更容易理解。例如，在常量的定义中就出现了关键字 let。Swift 的关键字总结如表 3-1 所示。

【示例 3-1】 以下将定义一个常量，并且输出常量的值。具体的操作步骤如下。

(1) 创建一个 Single View Application 模板类型的项目，将其命名为 3-1。

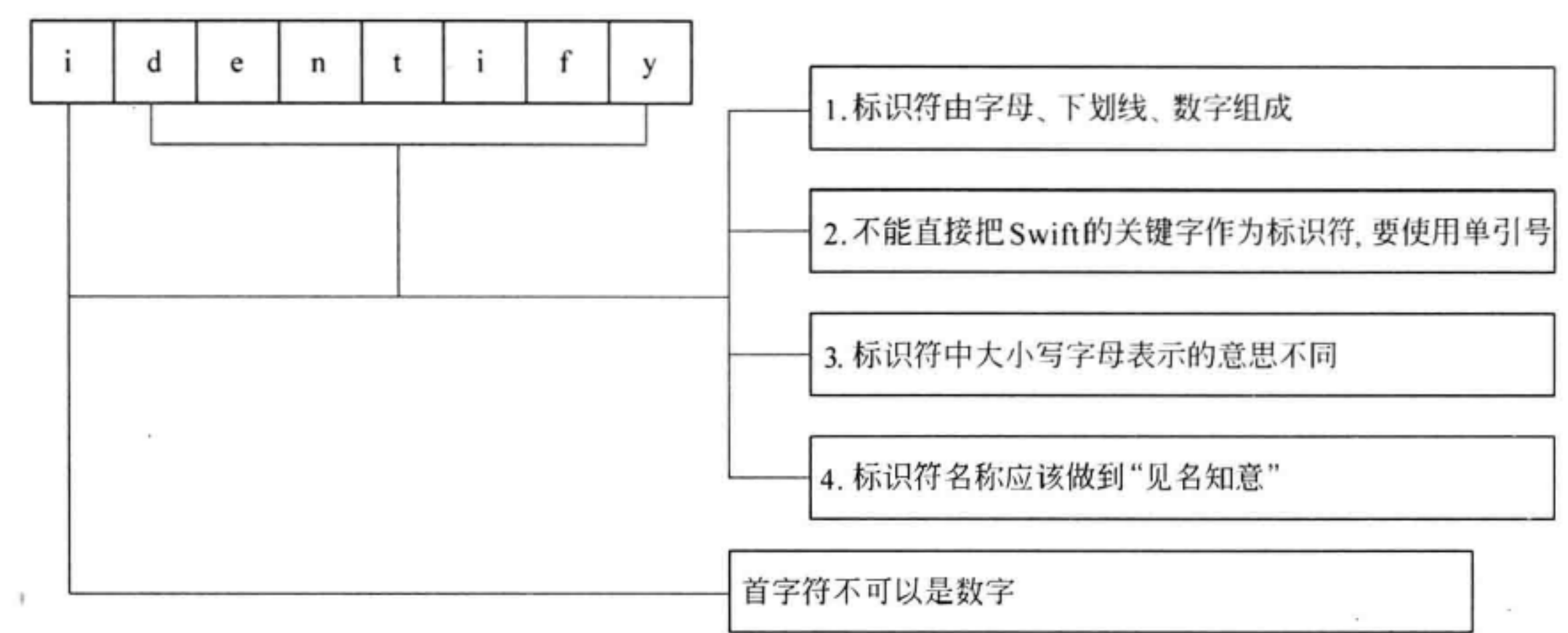


图 3.1 标识符的命名规则

表 3-1 关键字

用做声明的关键字				
class	deinit	enum	extension	func
import	init	let	protocol	static
struct	subscript	typealias	var	
用做语句的关键字				
break	case	continue	default	do
else	fallthrough	if	in	for
return	switch	where	while	
用做表达式和类型的关键字				
as	dynamicType	is	new	super
self	Self	Type	__COLUMN__	__FILE__
__FUNCTION__	__LINE__			
特定上下文中被保留的关键字				
associativity	didSet	get	infix	inout
left	mutating	none	nonmutating	operator
override	postfix	precedence	prefix	right
set	unowned	unowned(safe)	unowned(unsafe)	weak
willSet				

(2) 打开 ViewController.swift 文件，编写代码，实现常量的声明和定义，代码如下：

```
import UIKit
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let value=10 //常量的声明和定义
        println(value) //输出
    }
}
```

此时运行程序，会看到如下的结果：

```
10
```

在这里我们需要注意以下两点。

(1) 在程序中，我们使用了单行注释。注释是对程序的一种说明，并不影响程序的运行。一个恰当的注释是优秀代码的一部分。在 Swift 中注释分为 3 种：单行注释、多行注释，以及嵌套注释，以下就是对这 3 种注释的详细讲解。

单行注释是一次只有一行注释。单行注释以“//”开始，形式如下：

```
//xxxxxxxxxx
```

如图 3.2 所示就使用到了单行注释。

```
let value=10           //常量的声明和定义
println(value)         //输出
```

图 3.2 单行注释

多行注释就是一行或者多行叙述文字插入在一些注释分隔符中。这些注释分隔符以注释标记“/*”开始，以“*/”结束。形式如下：

```
/*xxxxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxx*/
```

如图 3.3 所示就使用到了多行注释。

```
/*此代码的功能是输出常量value的值，其中value的值为10，
println()是Swift自带的输出函数，
对于函数我们后面后面进行讲解*/
let value=10
println(value)
```

图 3.3 多行注释

嵌套注释就是在注释中再嵌套一些注释。如图 3.4 所示就使用到了嵌套注释。

```
/*此代码的功能是输出常量value的值，其中value的值为10
//println()是Swift自带的输出函数
对于函数我们后面后面进行讲解*/
let value=10
println(value)
```

图 3.4 嵌套注释

(2) 在后面的学习中，由于操作步骤以及编写代码的位置都是一样的，为了便于开发者的理解和不受干扰，我们只会给出编写的代码，如【示例 3-1】的代码，我们会写为：

```
let value=10
println(value)
```

3.1.2 变量

变量是用来指代一个可能变化的数据，正好和常量相反。在使用每个变量的时候，都需要先声明和定义，然后再使用。由于变量的声明和定义是同时进行的，所以将变量的声明和定义合并称为变量的定义。变量定义语法形式如下：

`var 变量名=值`

其中，var 是定义变量的关键字；变量名是变量的名称（注意，变量名必须符合标识符命名规范）。值表示变量被赋的值。

【示例 3-2】 以下将声明和定义一个变量，并输出变量所指代的值。代码如下：

```
var value=100 //声明并定义变量
println(value)
```

此时运行程序，会看到如下的结果：

100

3.2 数据类型

由于 Swift 支持类型推断（类型推断能使它自动推断出一个特定表达式的类型），所以在定义常量或变量时，可以不用特意指定数据类型。其实 Swift 语言和其他语言一样，也是可以在定义时指定数据类型的。其语法形式如下：

`let/var 常量/变量名:数据类型=值`


其中，数据类型包括整数类型和浮点类型等。数据类型是为了规范数据存储和运算方式所提供的语法特性。它决定了数据在计算机中的存储方式以及处理方式。在 Swift 中提供了很多的数据类型。本节就为开发者详细讲解这些数据类型。

3.2.1 整数类型

在讲解整数类型之前，首先需要知道整数。整数是没有小数部分的数字，如 40、50 等。整数类型就是为整数来指定尺寸范围的。由于整数根据位数可以分为 8 位、16 位、32 位和 64 位的形式，所以整数类型也分为了 8 位、16 位、32 位和 64 位这 4 种形式。又由于根据整数是否有符号可以将整数分为无符号整数和有符号整数，所以整数类型也可以分为无符号整数类型和有符号整数类型。Swift 中的整数类型和范围如表 3-2 所示。

表 3-2 整数类型和范围

整 数 类 型		范 围
8 位形式	UInt8 (uint8)	0~255
	Int8	-128~127
16 位形式	UInt16(uint16)	0~65535
	Int16	-32768~32767
32 位形式	UInt32 (uint32)	0~4294947295
	Int32	-2147483648~2147483647
64 位形式	UInt64 (uint64)	0~18446744073709551615
	Int64	-9223372036854775808~9223372036854775807


 **注意：**使用 U 开头的整数类型都是无符号的。

【示例 3-3】 以下将为两种变量指定数据类型。代码如下：

```
var value1:UInt8=100 //声明并定义一个类型为 UInt8 的变量
println(value1)
var value2:Int8 = -120 //声明并定义一个类型为 Int8 的变量
println(value2)
```

此时运行程序，会看到如下的结果：

```
100
-120
```

 **注意：** 由于位数不同，开发者可以通过整型类型的 min 和 max 属性查看这些类型的边界范围。

【示例 3-4】 以下将使用 min 和 max 属性获取 UInt8 和 Int8 的最大值和最小值，并输出。代码如下：

```
let minValue1=UInt8.min //获取最小值
let maxValue1=UInt8.max //获取最大值
println("UInt8 的最小值为 \(minValue1)")
println("UInt8 的最大值为 \(maxValue1)")
let minValue2=Int8.min //获取最小值
let maxValue2=Int8.max //获取最大值
println("Int8 的最小值为 \(minValue2)")
println("Int8 的最大值为 \(maxValue2)")
```

此时运行程序，会看到如下的结果：

```
UInt8 的最小值为 0
UInt8 的最大值为 255
Int8 的最小值为-128
Int8 的最大值为 127
```

在 Swift 除了以上提到的整数类型外，还提供了额外的整数类型 Int，一般开发者喜欢称其为整型，它具有和当前平台本地字一样的尺寸大小。例如，在一个 32 位的平台中，Int 的尺寸和 Int32 是一样的；在一个 64 位的平台中，Int 的尺寸和 Int64 是一样的。在代码中使用 Int 类型的整数值，可以有助于代码的一致性和互操作性。即使在 32 位的平台上，也可以存储-2147483648~2147483647 之间的任何值，并且对于很多的整数来说它的范围是足够大的。

以上所讲的是整数的有符号整型，下面讲解整数的无符号整型。在 Swift 中还提供了无符号的整型，即 UInt，它具有和当前平台本地字一样的尺寸大小。

□ 在一个 32 位的平台中，UInt 的尺寸和 UInt32 是一样的。

□ 在一个 64 位的平台中，UInt 的尺寸和 UInt64 是一样的。

在使用整型定义常量和变量时也是有两种类型的，即有符号整型以及无符号整型常量和变量。其定义两种整型常量的语法形式如下：

```
let 常量名:UInt=值
let 常量名:Int=值
```

定义两种整型变量的语法形式如下：

```
var 变量名:UInt=值
var 变量名:Int=值
```

【示例 3-5】 以下将使用 UInt 和 Int 类型分别声明一个常量和一个变量。代码如下：

```
let uvalue1:UInt=1000           //定义一个数据类型为无符号整型的常量
let uvalue2:Int = -800          //定义一个数据类型为有符号整型的常量
//输出常量值
println(uvalue1)
println(uvalue2)
var value1:UInt=200             //定义一个数据类型为无符号整型的变量
var value2:Int = -100           //定义一个数据类型为有符号整型的变量
//输出变量值
println(value1)
println(value2)
```

此时运行程序，会看到如下的结果：

```
1000
-800
200
-100
```

3.2.2 浮点类型

浮点数正好与整数相反，它是具有小数部分的数字，如 3.1415926、0.1 等。浮点类型表示的范围要比整数类型表示的范围更广，并且可以存储的数字也要比整数类型的更大或者更小。在 Swift 中提供了两种浮点数类型，如下：

❑ Double 表示 64 位的浮点数。当浮点值必须是非常大的或特别精确时，使用它。

❑ Float 表示 32 位的浮点数。当浮点值不需要像一样 64 位那样的精度时，使用它。

常量和变量也可以定义为一个浮点类型，这时就需要使用到 Double 或者 Float。在使用 Double 定义常量变量时的语法形式如下：

```
let 常量名:Double=值
var 变量名:Double=值
```

在使用 Float 定义常量和变量时的语法形式如下：

```
let 常量名:Float=值
var 变量名:Float=值
```

【示例 3-6】 下面使用 Double 和 Float 分别对常量和变量进行定义。代码如下：

```
let dvalue:Double=100.3         //定义一个数据类型为 Double 的常量
var fvalue:Float=100.3          //定义一个数据类型为 Float 的常量
//输出常量值
println(dvalue)
println(fvalue)
```

此时运行程序，会看到如下的结果：

```
100.3
100.300003051758
```


3.2.3 字符类型

在 Swift 中，提供了一种用于文本工作的类型即字符类型（Character），如"A"、"B"等。字符类型可以声明具有字符类型的常量和变量。它的定义方式如下：

```
let/var 常量名/变量名:Character=字符
```

【示例 3-7】 以下将使用 Character 字符类型分别定义常量和变量，并输出。代码如下：

```
let lcharacter:Character="A"           //定义一个数据类型为字符型的常量
println(lcharacter)
var vcharacter:Character="B"           //定义一个数据类型为字符型的变量
println(vcharacter)
```

此时运行程序，会看到如下的结果：

```
A
B
```

在 Swift 中还提供了一种用于处理文本的类型，即字符串类型（String）。所谓字符串，其实就是由一个或者多个字符组合而成的。如果想要将变量或者常量定义为字符串类型也是可以的，其语法形式如下：

```
let/var 常量名/变量名:String=字符串
```

【示例 3-8】 下面将使用 String 字符串类型分别定义具有字符串类型的常量和变量，并输出。代码如下：

```
let lstring:String="Hello, World"      //定义一个数据类型为字符串类型的常量
println(lstring)
var vstring:String="Swift"             //定义一个数据类型为字符串类型的变量
println(vstring)
```

此时运行程序，会看到如下的结果：

```
Hello, World
Swift
```

3.2.4 布尔类型

布尔类型（Boolean）表示布尔逻辑量。布尔类型又被叫做布尔（BOOL），在编程中可以将变量或者常量声明为布尔类型，语法形式如下：

```
let 常量名/变量名:Bool=布尔值
```

其中，布尔值是指代逻辑，因为它永远只有两个值，true 和 false。在 Swift 中提供了两种布尔常量值，true 和 false。

【示例 3-9】 以下将使用 Bool 布尔类型分别定义具有布尔类型的常量和变量，并输出。代码如下：


```
let lbool:Bool=false           //定义一个数据类型为布尔类型的常量
println(lbool)
var vbool:Bool=true           //定义一个数据类型为布尔类型的变量
println(vbool)
```

此时运行程序，会看到如下的结果：

```
false
true
```

3.2.5 可选类型

可选类型用来判断值是否存在。如果值存在就会输出，如果不存在，就会返回一个 nil（nil 是一个特定类型的空值。任何类型的可选变量都可以被设置为 nil），它是 Swift 专用类型。可选类型常量和变量的定义是使用问号实现的，其语法形式如下：

Let/var 常量名/变量名:数据类型?

【示例 3-10】 以下将使用? 定义一个可选类型的常量和变量。代码如下：

```
let a:Int?=100                 //可选常量
println(a)
var b:Int?                     //可选变量
println(b)
var c:Int?=nil                 //可选变量
println(b)
```

此时运行程序，会看到如下的结果：

```
Optional(100)
nil
nil
```

3.2.6 类型别名

类型别名就是为现有类型定义的替代名称。类型别名可以帮助开发者使用更符合上下文语境的名字来指代一个已存在的类型。对于类型别名的定义，可以使用 typealias 关键字实现。其语法形式如下：

typealias 类型别名=数据类型名称

其中，typealias 是关键字；类型别名表示标识符；数据类型名称表示整型和字符型等。

【示例 3-11】 下面使用 typealias 关键字定义一个类型别名，并使用 max 输出此类型的最大值范围。程序代码如下：

```
typealias AudioSample = Int
var maxAudioSample=AudioSample.max
println(maxAudioSample)
```

此时运行程序，会看到如下的结果：

```
9223372036854775807
```

3.3 值的表示——字面值

字面值和其他的编程语言中提供的用法一样，以人们易于阅读的格式表示的固定值。一般字面值的用法是非常直观的。在 Swift 中提供了整型字面值、浮点型字面值、字符型字面值、字符串字面值，以及布尔型字面值等。下面就为各位开发者做一个简单的介绍。

3.3.1 整型字面值

整型字面值可以写为以下 4 种形式：

- ❑ 十进制数字，没有前缀。
- ❑ 二进制数，用前缀 0b 表示。
- ❑ 八进制数，用前缀 0 表示。
- ❑ 十六进制数，用前缀 0x 表示。

【示例 3-12】 下面就使用 4 种形式为各位开发者显示值为 17 的整型字面值，代码如下：

```
let decimalInteger = 17
let binaryInteger = 0b10001           //采用二进制数表示 17
let octalInteger = 017                 //采用八进制数表示 17
let hexadecimalInteger = 0x11         //采用十六进制数表示 17
println(decimalInteger)
println(binaryInteger)
println(octalInteger)
println(hexadecimalInteger )
```

此时运行程序，会看到如下的结果：

```
17
17
17
17
```

3.3.2 浮点型字面值

浮点类型的字面值可以使用十进制数（不带前缀），或者十六进制数（带有前缀 0x）表示，并且它们必须在小数点的两侧。

【示例 3-13】 以下将使用浮点型字面值显示 12.1875。代码如下：

```
let value = 12.1875
println(value)
```

此时运行程序，会看到如下的结果：

```
12.1875
```

注意：浮点数也可以使用科学计数法表示。其中，使用大写或者小写的 e 表示十进制的浮点数，使用大写或者小写的 p 表示十六进制的浮点数，其语法形式如下：

n.ne+/-P//十进制的浮点数

n.np+/-p//十六进制的浮点数

其中，p 表示小数点移动的位数；+表示小数点向右移，-表示小数点向左移。例如：

//十进制的浮点数

1.25e2//表示 1.25*10² 或者 125.0

1.25e-2//表示 1.25*10⁻² 或者 0.0125

//十六进制的浮点数

0xFp2//表示 15*2² 或者 60

0xFp-2//表示 15*2⁻² 或者 3.75

3.3.3 字符型字面值

字符型字面值通常使用双引号表示，如"A"、"B"等。

【示例 3-14】 以下将使用字符型字面值为一个字符类型的常量赋值。代码如下：

```
let char = "A"
println(char)
```

此时运行程序，会看到以下的结果：

A

3.3.4 字符串型字面值

字符串字面值是由一对双引号包围的固定顺序的文本字符。

【示例 3-15】 以下将使用字符串型字面值为一个字符串类型的常量赋值。代码如下：

```
let string:String="Hello,Swift"
println(string)
```

此时运行程序，会看到以下的结果：

Hello,Swift


 **注意：**使用字符串字面值对于大部分打印字符是有效的。但是对于一些非打印字符就无法直接表示它们了，如回车等。所以 Swift 提供了一些特定的字符去表示它们——转义序列。其中，转义序列是由反斜杠和字符组合在一起的字符串。转义字符的种类以及功能如表 3-3 所示。

表 3-3 转义字符的种类及功能

转 义 序 列	功 能
\0	空
\\	反斜杠
\t	水平制表符
\n	换行
\r	回车
\"	双引号

【示例 3-16】 以下将使用\n 转义序列让字符串“Hello”进行换行。代码如下：

```
let my="\nHello"
println(my)
```

此时运行程序，会看到以下的结果：

```
Hello //这里有一个换行
```

3.3.5 布尔型字面值

布尔类型的字面值只有 true 和 false。

【示例 3-17】 以下将使用布尔类型的字面值为一个常量赋值。代码如下：

```
let value=true
println(value)
```

此时运行程序，会看到以下的结果：

```
true
```

3.3.6 元组型字面值

元组就是将多个值放到一起，并组合成一个元素。开发者可以将自己声明的常量或者变量定义为元组类型，其语法形式如下：

```
let/var 常量名/变量名=元组类型的字面值
```

其中，元组类型的字面值需要使用括号括起来，其语法形式如下：

```
(值 1, 值 2, 值 3, 值 4, ……)
```

其中，值可以是任意的数据类型，例如：


```
(1, 10.22, "Hello World")
```

【示例 3-18】 以下将声明的变量定义为元组类型。代码如下：

```
var value=(403, "Not Found") //元组类型的变量
println(value)
```

此时运行程序，会看到以下的结果：

```
(403, Not Found)
```

 **注意：** 可以将一个变量或者常量定义为元组，还可以同时将多个变量或者常量定义为元组，其语法形式如下：

```
var(变量名 1, 变量名 2, 变量名 3, ……)=元组类型的字面值
let(常量名 1, 常量名 2, 常量名 3, ……)=元组类型的字面值
```

【示例 3-19】 以下同时将多个变量定义为元组类型。代码如下：

```
var (value1, value2, value3)=(1, "Hello", 5) //元组类型的多个变量
```

```
println(value1)
println(value2)
println(value3)
```

此时运行程序，会看到以下的结果：

```
1
Hello
5
```

3.4 运 算 符

在 Swift 中，提供了丰富的运算符。它们在程序中发挥了重要的作用。其功能是执行程序代码运算，会针对一个或者一个以上的操作数项目来进行运算。本节将主要讲解这些运算符。

3.4.1 元的介绍

元表示运算符所使用的目标数值个数（即操作数）。根据数值个数的不同，运算符分为一元运算符、二元运算符、三元运算符。它们的详细说明如表 3-4 所示。

表 3-4 常用术语总结

术 语	说 明
一元运算符	它对一个目标进行操作。一元运算符分为一元前缀运算符和一元后缀运算符。其中，一元前缀运算符出现在目标的前面（例如 -b）；一元后缀运算符出现在目标的后面（例如 b--）
二元运算符	它对两个目标进行操作，并且是中缀（即在两个操作数之间）
三元运算符	它对三个目标进行操作。与 C 语言一样，Swift 也只有一个三元运算符，即三元跳转运算符（a?b:c）

3.4.2 赋值运算符

赋值运算符其实在前面几节的代码中见到过了，其功能就是给变量或者常量进行赋值。赋值运算符使用=来实现的，语法形式如下：

```
操作数=操作数;
```

在程序中指定赋值是双目的。

【示例 3-20】 以下将使用单一的赋值运算符实现赋值。代码如下：

```
var a:Int
a=100 //赋值
println(a)
```

此时运行程序，会看到如下的结果：

```
100
```

3.4.3 一元加运算符

在操作数前加一个+号，此+号就被叫做一元加运算符，它基本上没有什么作用，只是为了对齐代码。由一元加运算符连接起来的式子被称为一元加表达式。其语法形式如下：

+操作数

【示例 3-21】 以下在变量的前面使用了一元加运算符。代码如下：

```
var a=8
var b = +a
println(a)
```

此时运行程序，会看到以下的结果：

8

3.4.4 一元减运算符

在操作数之前加一个-号，此-号就被叫做一元减运算符。它的作用是将正数变为负数，将负数变为正数。由一元减运算符连接起来的式子被称为一元减表达式。其语法形式如下：

-操作数

【示例 3-22】 以下使用一元减运算符将数值 8 变为负数，再将变为负数的 8 变为正数。代码如下：

```
var a=8
println("a=\(a)")
var b = -a
println("b=\(b)")
var c = -b
println("c=\(c)")
```

此时运行程序，会看到以下的结果：

a=8
b=-8
c=8

3.4.5 算术运算符

算术运算符就是我们常用的加、减、乘、除、取余运算。算术运算均为双目运算，我们将 Swift 中的算术运算符为大家做一个总结，如表 3-5 所示。

表 3-5 算术运算符

运算符名称	符 号	功 能	结 合 性
加法运算符	+	将两个数相加	左到右
减法运算符	-	将两个数相减	

续表			
运算符名称	符 号	功 能	结 合 性
乘法运算符	*	将两个数相乘	左到右
除法运算符	/	将两个数相除	
取余运算符	%	取两个数相除后的余数	

由算术运算符构成的表达式称为算术表达式。算术表达式的语法如下：

操作数 算数运算符 操作数


其中，操作数一般是整数和浮点数。操作数不仅可以为整数，还可以为负数。

【示例 3-23】 以下将使用算术运算符实现一些计算。代码如下：

```
var a=10
var b=5
var c=a+b           //加法运算
var d=a-b           //减法运算
var e=a*b           //乘法运算
var f=a/b           //除法运算
var g=a%3           //取余运算
//输出结果
println(c)
println(d)
println(e)
println(f)
println(g)
```

此时运行程序，会看到以下的结果：

```
15
5
50
2
1
```

 **注意：**当有多个算术运算符时，需要注意它们的运算优先级别。所谓优先级其实就是谁先执行，谁后执行。其中，*、/的优先级最高，其次是%，最后是+、-。

3.4.6 自增、自减运算符

自增、自减运算符其实就是使变量的值自加或自减 1。自增、自减运算符是单目运算。以下将对这两个运算符进行详细讲解。

1. 自增运算符

自增运算符其实就是使变量的值自加 1，它可以写为++。自增表达式的形式有两种，一种是前缀自增，另一种是后缀自增。语法形式如下：

```
++变量名           //前缀自加
变量名++           //后缀自加
```

【示例 3-24】 以下将使用自增运算符实现运算。代码如下：

```
var value=9
println(value)
println(++value)           //前缀自加
println(value)
println(value++)          //后缀自加
println(value)
```

此时运行程序，会看到以下的结果：

```
9
10
10
10
11
```

⚠注意：在此程序中，我们需要注意，`++value` 和 `value++` 所产生的结果是不一样的，`++value` 是先执行+1，再输出；而 `value++` 则是先输出，后执行+1。`value++` 也可以写为 `value=value+1`。

2. 自减运算符

自减运算符其实就是使变量的值自减 1，它可以写为`--`。自增表达式的形式有两种，一种是前缀自减，另一种是后缀自减，语法形式如下：

```
--变量名           //前缀自减
变量名--           //后缀自减
```

【示例 3-25】 以下将使用自减运算符实现运算。代码如下：

```
var value=10
println(value)
println(--value)           //前缀自减
println(value)
println(value--)          //后缀自减
println(value)
```

此时运行程序，会看到以下的结果：

```
10
9
9
9
8
```

⚠注意：在此程序中，我们需要注意，`--value` 和 `value--` 所产生的结果是不一样的。`--value` 是先执行-1，再输出；而 `value--` 则是先输出，后执行-1。`value--` 也可以写为 `value=value-1`。

3.4.7 比较运算符

比较运算符一般用在比较运算中，即实现两个操作数的大小比较。在 Swift 中，提供了 6 种比较运算符，如表 3-6 所示。

表 3-6 比较运算符

运 算 符	运算符名称	功 能	实 例	结 果
<	小于	若 a<b, 结果为 true, 否则为 false	2<3	true
<=	小于等于	若 a<=b, 结果为 true, 否则为 false	7<=3	false
>	大于	若 a>b, 结果为 true, 否则为 false	7>3	true
>=	大于等于	若 a>=b, 结果为 true, 否则为 false	3>=3	true
==	等于	若 a==b, 结果为 true, 否则为 false	7==3	false
!=	不等于	若 a!=b, 结果为 true, 否则为 false	7!=3	true

比较运算符通常用在条件语句中（对于条件语句会在后面的章节中介绍）。使用比较运算符连接起来的式子称为比较表达式。其语法形式如下：

表达式 比较运算符 表达式

⚠注意：表达式可以是一个运算符，比较表达式返回的类型为 Bool（布尔类型）。

【示例 3-26】 下面就使用比较运算符>、<、!=、==对 10 和 5 进行比较。代码如下：

```
var a=10
var b=5
var c=a>b //10 是否大于 5
println(c)
var d=a<b //10 是否小于 5
println(d)
var e = a != b //10 是否不等于 5
println(e)
var f = a == b //10 是否等于 5
println(f)
```

此时运行程序，会看到如下的结果：

```
true
false
true
false
```

3.4.8 逻辑运算符

在很多的编程语言中，一个功能往往需要满足多个条件才可以执行。这时就需要将这多个条件进行组合。逻辑运算符的功能就可以把这多个条件进行组合，从而实现更复杂的表达式。使用逻辑运算符连接起来的式子被称为逻辑表达式。其语法形式如下：

条件表达式 逻辑运算符 条件表达式

其中，逻辑表达式返回的值是 Bool（布尔值）。在 Swift 语言中包括 3 种逻辑运算符，如表 3-7 所示。

表 3-7 逻辑运算符

逻辑运算符	名 称	使 用 形 式	功 能
&&	逻辑与	(表达式 1)&&(表达式 2)&&...	参与运算的表达式都为真时，结果才为真

续表

逻辑运算符	名 称	使 用 形 式	功 能
	逻辑或	(表达式 1) (表达式 2) ...	参与运算的表达式中只要有一个表达式为真，结果就为真
!	逻辑非	!表达式	参与运算的表达式为真，结果就为假，表达式为假，结果就为真

【示例 3-27】 以下将使用 3 种逻辑运算符实现运算。代码如下：

```
//使用逻辑与
let a = 2<5 && 7>5
println(a)
let b = 2>5 && 7>5
println(b)
//使用逻辑或
let c = 2>5 || 7>5
println(c)
let d = 2>5 || 7<5
println(d)
let e = 2<5 || 7<5
println(e)
//使用逻辑非
let f = !(7<10)
println(f)
let g = !(7>10)
println(g)
```

此时运行程序，会看到如下的结果：

```
true
false
true
false
true
false
true
```

3.4.9 位运算符

位是用以描述计算机数据量的最小单位。二进制系统中，每个 0 或 1 就是一个位。位运算是指按二进制进行的运算。在 Swift 中有专门对位运算使用的运算符——位运算符。位运算符通常在诸如图像处理和创建设备驱动等底层开发中使用。使用它可以单独操作数据结构中原始数据的比特位。位运算符的总结如表 3-8 所示。

表 3-8 位运算符

位运算符符号	位运算符名称	作 用
&	按位与	两个相应的二进制位都为 1，则该位为 1，否则为 0
	按位或	两个相应的二进制位中只有一个为 1，则该位为 1
^	按位异或	两个相应的二进制位值相同则为 0，否则为 1
~	取反	将二进制数按位取反，即 0 变 1、1 变 0
<<	左移	将一个数的各二进制位全部左移 N 位，右补 0
>>	右移	将一个数的各二进制位全部右移 N 位，对于无符号位，高位补 0

【示例 3-28】 以下将使用位运算符来实现操作数的位运算。代码如下：

```
var a:UInt8=0b11111100
var b:UInt8=0b00111111
var c=a&b //按位与运算
println(c)
var d=a|b //按位或运算
println(d)
var e=a^b //按位异或运算
println(e)
var f = ~a //取反运算
println(f)
var g=a<<4 //左移
println(g)
var h=b>>4 //右移
println(h)
```

此时运行程序，会看到如下的结果：

```
60
255
195
3
192
3
```

3.4.10 复合运算符

在多数语言中，都有复合赋值运算符，在 Swift 语言中也不例外。复合运算符是由赋值运算符和其他的一些运算符组合起来的。其中，复合赋值运算符的种类、使用方法，以及功能如表 3-9 所示。

表 3-9 复合赋值运算符

符 号	使 用 方 法	等 效 形 式	功 能
<code>*</code>	<code>a*=b</code>	<code>a=a*b</code>	乘后赋值
<code>/</code>	<code>a/=b</code>	<code>a=a/b</code>	除后赋值
<code>%</code>	<code>a%=b</code>	<code>a=a%b</code>	取余后赋值
<code>+</code>	<code>a+=b</code>	<code>a=a+b</code>	加后赋值
<code>-</code>	<code>a-=b</code>	<code>a=a-b</code>	减后赋值
<code><<=</code>	<code>a<<=b</code>	<code>a=a<<b</code>	左移后赋值
<code>>>=</code>	<code>a>>=b</code>	<code>a=a>>b</code>	右移后赋值
<code>&=</code>	<code>a&=b</code>	<code>a=a&b</code>	按位与后赋值
<code>^=</code>	<code>a^=b</code>	<code>a=a^b</code>	按位异或后赋值
<code> =</code>	<code>a =b</code>	<code>a=a b</code>	按位或后赋值

由这些复合赋值运算符连接起来的式子被称为复合赋值表达式。其语法形式如下：

```
变量 复合赋值运算符 表达式
```

【示例 3-29】 以下将使用复合赋值运算符实现运算。代码如下：

```
var a=1
```



```
println(a)
a+=10
println(a)
a*=10
println(a)
a-=10
println(a)
a/=3
println(a)
```

此时运行程序，会看到如下的结果：

```
1
11
110
100
33
```

3.4.11 求字节运算符

由于不同的计算机所支持的数据类型长度也是不一样的，所以就提供了一个用来计算数据类型所占字节数的运算符——sizeof。由 sizeof 运算符连接起来的式子被称为求字节表达式。其语法形式如下：

```
sizeof(数据类型)
```

【示例 3-30】 以下将使用 sizeof 求字节运算符对整型、浮点型、字符，以及字符串所占的字节数进行获取。代码如下：

```
let a=sizeof(Int)
println(a)
let b=sizeof(Float)
println(b)
let c=sizeof(Character)
println(c)
let d=sizeof(String)
println(d)
```

此时运行程序，会看到如下的结果：

```
8
4
9
24
```

3.4.12 强制解析运算符

可选类型其实是一个枚举类型，里面包含了 None 和 Some 两个类型。其实，所谓的 nil 就是 Optional.None，非 nil 就是 Optional.Some，然后通过 Some(T) 包装 (wrap) 原始值，所以在 3.2.5 节中看到当可选类型有值时，输出的值类似于以下的形式：

```
Optional(原始值)
```

所以在使用可选类型进行运算时，需要拆包，即从枚举类型中取出原始值，拆包就需

要使用到强制解析运算符，即！，它的使用形式如下：

可选类型的变量名/常量名！

【示例 3-31】 以下将使用强制解析运算符实现可选类型的拆包，以及实现加法赋值运算。代码如下：

```
var a:Int?=10
println(a)
println(a!) //拆包
a! += 20 //加法赋值运算
println(a!)
```

此时运行程序，会看到如下的结果：

```
Optional(10)
10
30
```

3.4.13 区间运算符

在 Swift 中提供了两种可以方便的表达区间值的运算符：一种是闭区间运算符，另一种是半闭区间运算符。

1. 闭区间运算符

闭区间运算符为...。由闭区间运算符连接起来的式子被称为闭区间表达式。其语法形式如下：

操作数 1...操作数 2

其中，区间从操作数 1～操作数 2，并且包括操作数 1 和操作数 2。操作数 1 必须要小于操作数 2。

【示例 3-32】 以下将使用闭区间运算符实现对 1～5 这 5 个数字的输出。代码如下：

```
//遍历，输出 1～5 这 5 个数字
for index in 1...5 {
    println(index)
}
```

此时运行程序，会看到如下的结果：

```
1
2
3
4
5
```

2. 半闭区间运算符

半闭区间运算符为.. $<$ 。由半闭区间运算符连接起来的式子被称为半闭区间表达式。其语法形式如下：

操作数 1.. $<$ 操作数 2

其中，范围从操作数 1~操作数 2，但是只包括操作数 1，不包括操作数 2。

【示例 3-33】 下面就使用半封闭范围运算符实现对 1~4 这 4 个数字的输出。代码如下：

```
//遍历，输出 1~4 这 4 个数字
for index in 1...<5 {
    println(index)
}
```

此时运行程序，会看到如下的结果：

```
1
2
3
4
```

3.4.14 溢出运算符

一般，为一个整型变量或常量赋值时都不会超出它的承载范围。当超出时，Swift 也不会让程序通过，它会报错。如果开发者有意进行溢出操作，可以使用溢出运算符。在 Swift 中提供了 5 种针对整型的溢出运算符，如表 3-10 所示。

表 3-10 溢出运算符

溢出运算符	说 明
&+	溢出加法
&-	溢出减法
&*	溢出乘法
&/	溢出除法
&%	溢出取余

使用溢出运算符连接起来的式子被称为溢出运算符，其语法形式如下：

```
操作数 1 溢出运算符 操作数 2
```

⚠注意：如果溢出运算符为&+或者&*时，操作数 1 需要为最大值；如果溢出运算符为&-时，操作数 1 需要为最小值；如果溢出运算符为&/或者&%时，操作数 2 需要为 0。

【示例 3-34】 以下将使用溢出运算符实现运算。代码如下：

```
var minvalue=UInt8.min
minvalue = minvalue &- 1
println(minvalue)
var maxvalue=Int8.max
maxvalue = maxvalue &+ 1
println(maxvalue)
var x=0
var y=10 &/ x
println(y)
```

此时运行程序，会看到如下的结果：

```
255
-128
0
```

3.5 类型转换

在使用运算符进行运算时，经常会遇到左右两个操作数不是同一类型的问题。这时，就需要使用到类型转换。在 Swift 中，所有的类型转换都为显示转换。以下将对整数的转换、整数和浮点数转换进行讲解。

3.5.1 整数的转换

整数类型分为 8 种。当它们中的两种或者两种以上出现在同一个表达式中时，就需要进行类型转换。其转换的语法形式如下：

整数的数据类型 (整数类型的变量/常量)

其中，整数的数据类型有 UInt8、UInt16、UInt32、UInt64、Int8、Int16、Int32、Int64。

【示例 3-35】 以下将 UInt8 类型的数据转换为 UInt16 类型的数据。代码如下：

```
let a:UInt8=200
let b:UInt16=2000
let c=UInt16(a)+b           //转换
println(c)
```

此时运行程序，会看到如下的结果：

```
2200
```

3.5.2 整数与浮点数的转换

整数除了在整数类型之间进行转换外，还可以和浮点数进行转换。首先来了解整数转换为浮点数，其语法形式如下：

浮点数类型 (整数的变量/常量)

其中，浮点数类型有两种：一种是 Float；另一种是 Double。

【示例 3-36】 以下将实现整数和浮点数的转换。代码如下：

```
let a=3
let b=10.555555
let c=Double(a)+b           //将整数转换为 Double 类型的数据
println(c)
let d:Float=10.22
let e=Float(a)+d            //将整数转为 Float 类型的数据
println(e)
```

此时运行程序，会看到如下的结果：

```
13.555555
13.2200002670288
```


3.6 字符串

字符串其实在前面章节中已经讲解过了。字符串实际就是一个字符序列，如"Hello World"和"Swift"等。在 Swift 中使用关键字 String 定义字符串。本节将讲解一些有关字符串的内容。

3.6.1 字符串的初始化

字符串初始化的方式有很多，最常使用到的就是使用一个字符串去初始化另一个字符串。

【示例 3-37】 以下将使用字符串"Hello World"去初始化字符串 a，代码如下：

```
var a:String="Hello World"
println(a)
```

此时运行程序，会看到如下的结果：

```
Hello World
```

3.6.2 字符串的操作

表 3-11 中总结了字符串常用的一些操作。

表 3-11 字符串常用操作

属 性	
uppercaseString	将字符串中所有的小写字符转换为大写字符
lowercaseString	将字符串中所有的大写字符转换为小写字符
isEmpty	判断字符串是否为空
方 法	
countElements	获取字符串中字符的个数
hasPrefix	判断字符串是否以某一字符串为前缀
hasSuffix	判断字符串是否以某一字符串为后缀
运 算 符	
+	将字符串和字符（或者是字符串）组合起来
+=	将字符串和字符（或者是字符串）进行组合
==	判断两个字符串是否相等

【示例 3-38】 以下将使用 countElements 方法获取字符串中字符的个数，然后使用 uppercaseString 属性将字符串中所有的字符变为大写。代码如下：

```
var a:String="Hello World"
println(countElements(a))           //获取字符串中字符的个数
println(a.uppercaseString)         //将字符串中的字符转为大写
```

此时运行程序，会看到以下的结果：

```
11
HELLO WORLD
```

3.7 集合类型

使用集合类型可以存储多个数据。在 Swift 中有两种集合类型，即数组和字典。本节将对这两种集合类型进行讲解。

3.7.1 数组

数组是用来存储相同类型的序列化列表。相同的值可以在数组的不同位置出现多次。在 Swift 语言中，数组自身储存的数据类型是确定的，所以它很安全。数组同样也是一种变量，只是所指代的值比较特殊而已。所以，数组在使用之前，必须进行声明和定义。声明数组的完整写法形式如下：

```
Array<SomeType>
```

或者：

```
[SomeType]
```

其中，`SomeType` 为数据类型。虽然这两种方法在功能上是相同的。但是更希望开发者使用后者，而且它会一直用于本书。一般数组的声明和定义是放在一起进行的，其语法形式形式如下：

```
let 常量数组名: [SomeType]=内容
var 变量数组名: [SomeType]=内容           //内容可写可不写
```

其中，`[SomeType]`是可以省略不写的，Swift 会自动推断其类型。内容的书写如下：

```
[内容 1, 内容 2, .....]
```

或者：

```
[]
```

【示例 3-39】 以下将对数组进行初始化。代码如下：

```
var a=[1,2,"Hello"]
println(a)
var b=[]
println(b)
var c:[Int]=[1,2,3]
println(c)
```

此时运行程序，会看到如下的结果：

```
(
    1,
    2,
    Hello
)
```

```
(
)
[1, 2, 3]
```

⚠注意：由于使用到的初始化的方式不同，所有输出的内容也不同。

在数组中我们可以进行很多的操作，如判断、插入和删除等。表 3-12 对数组的常用操作进行了总结。

表 3-12 数组常用操作

属 性	
count	读取数组的长度
isEmpty	判断数组是否为空
方 法	
copy	复制数组
append	添加元素
insert	在特定索引值位置处插入一个值（元素）
removeLast	删除数组中最后一个值
removeAtIndex	通过索引值将数组中任意位置的元素进行删除
removeAll	数组的所有元素都删除
运 算 符	
===	判断给定的两个数组是否共用相同的存储空间或元素
!==	判断给定的两个数组是否没有共用相同的存储空间或元素
+=	将一个元素添加到数组的末尾
[]	读取值或者是修改值

【示例 3-40】 以下将对数组 a 进行一些操作，如添加和删除。代码如下：

```
var a=[1,2,3,4,5,6]
println("a 数组的元素个数: \(a.count)")           //获取数组的个数
a += [7]                                           //添加元素
println(a)
a.removeLast()                                     //删除最后一个元素
println(a)
```

此时运行程序，会看到如下的结果：

```
a 数组的元素个数: 6
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6]
```

3.7.2 字典

字典是用于存储同一类型不同值的集合。它存储的每个元素包含一个键（Key）和一个值。其中的值都和键相对应。在 Swift 中，对于一个特定的字典，它所能储存的键和值都是确定的，无论是明确声明的类型还是隐式推断的类型。字典在使用之前，必须进行声明和定义。声明字典的完整写法形式如下：

```
Dictionary<KeyType,ValueType>
```

其中，KeyType 表示键的数据类型；ValueType 表示值的数据类型。需要注意的是，

KeyType 必须是可哈希的（hashable）——就是提供一个形式让它们自身是独立识别的。Swift 的所有基础类型（例如字符串（String）、整型（Int）、双精度（Double）和布尔（Bool））默认都是可哈希的（hashable）。一般字典的声明和定义是放在一起进行的，其语法形式形式：

```
let 常量字典名: Dictionary<KeyType,ValueType>=内容
var 变量字典名: Dictionary<KeyType,ValueType>=内容
```

其中，Dictionary<KeyType,ValueType>是可以省略的。内容其实就是一些字典字面量，它已被包含一个或者多个键值对的字典数值。一个键值对是一个键和值的组合。键和值使用冒号分隔，而多个键值对使用逗号分隔。其键值对语法形式如下：

```
key:value
```

其中，参数 key 表示键，value 表示值。字典字面量就是有键值对和一对[]组合。字典字面量可以只有一个键值对。这时的字典字面量语法形式如下：

```
[key:value]
```

当字典字面量有多个键值对时，其语法形式如下：

```
[key1:value1,key2:value2,key3:value3……]
```

【示例 3-41】 以下将对字典进行声明和定义，代码如下：

```
var dic=[1:"a",2:"b",3:"3"]
println(dic)
```

此时运行程序，会看到如下的结果：

```
[1: a, 2: b, 3: 3]
```

在字典中也可以进行很多的操作，如判断、插入和删除等。表 3-13 对字典的常用操作进行了总结。

表 3-13 字典常用操作	
属 性	
count	读取字典的长度
方 法	
updateValue	对字典中键关联的值进行修改
removeValueForKey()	将字典中指定键关联的值删除
removeAll	字典的所有元素都删除
运 算 符	
[]	读取键的值
	添加元素
	修改元素

【示例 3-42】 以下将对字典 a 进行一些操作，如添加和删除等。代码如下：

```
var a=[1:"a",2:"b",3:"c"]
println(a)
a[4]="d" //添加
println(a)
a.removeValueForKey(3) //删除
```

```
println(a)
var value=a[2]           //读取
println(value)
```

此时运行程序，会看到如下的结果：

```
[1: a, 2: b, 3: c]
[1: a, 2: b, 3: c, 4: d]
[1: a, 2: b, 4: d]
Optional("b")
```

3.8 程序控制结构

3.8.1 顺序结构

顺序结构的执行顺序是自上而下的，它是程序控制结构中最简单的一种。它的执行流程图如图 3.5 所示。

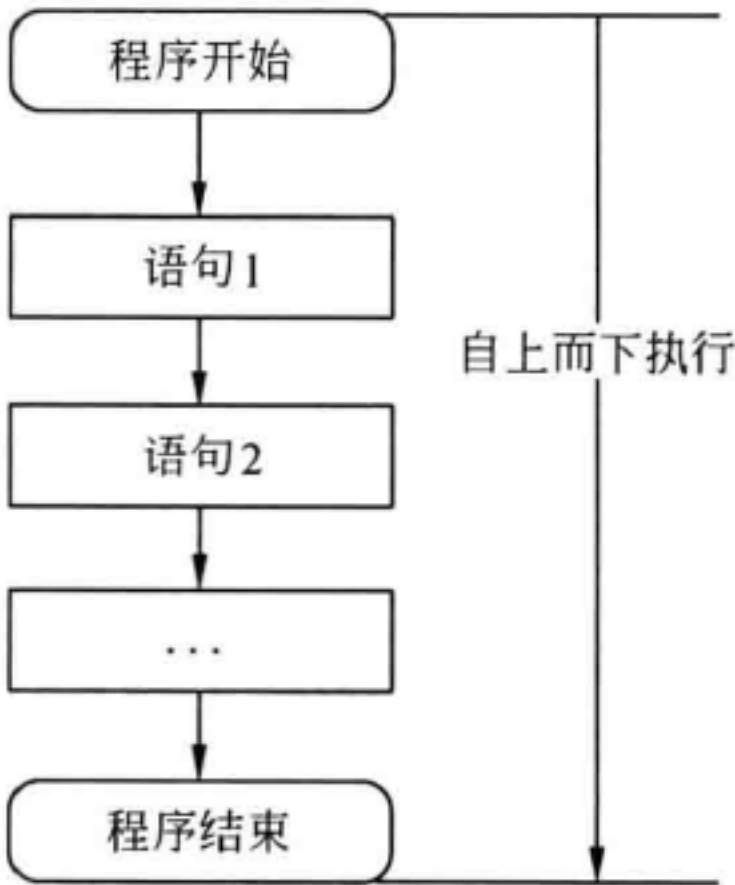


图 3.5 流程图

【示例 3-43】 以下将实现求三角形的面积。代码如下：

```
var a=10
println("底为\ (a) ")
var b=10;
println("高为\ (b) ")
var c=a*b/2;
println("三角形的面积为\ (c) ")
```

此时运行程序，会看到如下的结果：

```
底为 10
高为 10
三角形的面积为 50
```

3.8.2 选择结构

选择结构会判断给定的条件，根据判断的结果来选择执行程序。Swift 语言提供 3 种选

择结构的语句：条件运算符、if 语句和 switch 语句。以下就是对这 3 种语句的详细介绍。

1. 条件运算符

条件运算符是最简单的选择结构。它的语法形式如下：

```
表达式 1 ? 表达式 2 : 表达式 3
```

其中，表达式 1 通常是由关系运算符组成的表达式，也就是关系表达式。如果表达式 1 的结果为真，就执行表达式 2，表达式 2 的结果就是该运算的结果；如果表达式 1 的结果为假，那么，就会执行表达式 3，表达式 3 的结果就为该运算的结果。

【示例 3-44】 以下将通过使用条件运算符，实现最大值的获取。代码如下：

```
var a=10
var b=5
var c = a > b ? a : b
println(c)
```

此时运行程序，会看到如下的结果：

```
10
```

2. if 语句

if 语句是用来判断所给定的条件是否满足，根据判断的结果执行不同的语句。if 语句有 3 种用法。

(1) if 语句

if 语句是最基本的语句，它只能判断一种情况。if 语句的语法形式如下：

```
if(表达式)
    语句
```

其中，表达式为真时，执行语句；否则执行 if 之外的语句，在这里需要注意，语句可以是一条语句，也可以是多条语句。它的执行流程图如图 3.6 所示。

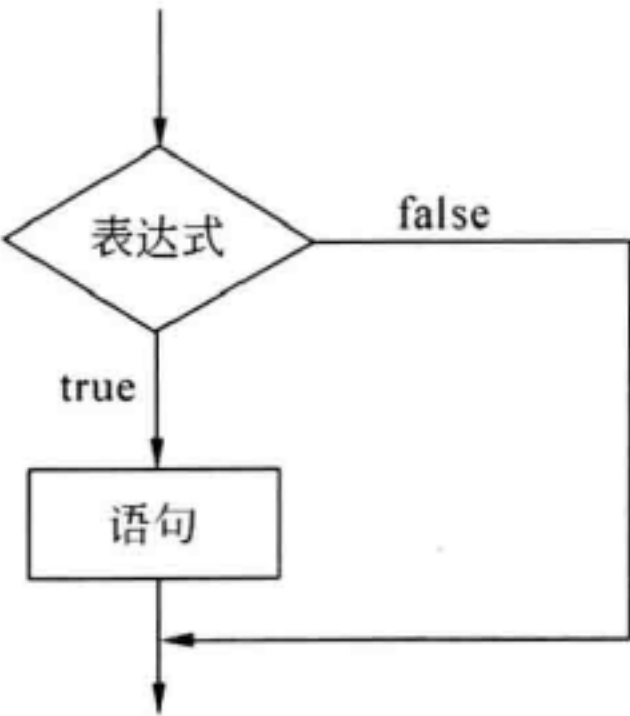


图 3.6 流程图

【示例 3-45】 以下将使用 if 语句来判断变量 a 和 b 中的最大值。代码如下：

```
var a=5
var b=10
//判断
```



```
if(a<b){
    println(b)
}
```

此时运行程序，会看到如下的结果：

```
10
```

(2) if...else 语句

if...else 语句的语法形式如下：

```
if(表达式)
    语句 1
else
    语句 2
```

其中，当表达式的条件成立时，就执行 if 后面的语句 1；当表达式的条件不成立时，就执行语句 2。在这里需要注意，语句 1 和语句 2，可以是一条语句，也可以是多条语句。它的执行流程图如图 3.7 所示。

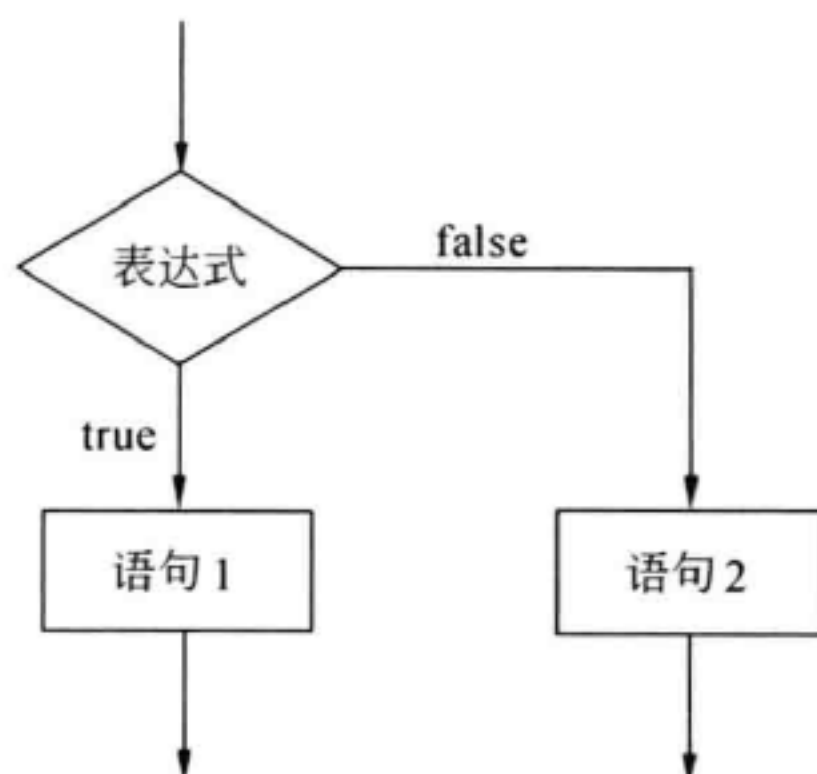


图 3.7 流程图

【示例 3-46】 以下将使用 if...else 语句来判断变量 a 和 b 中的最大值。代码如下：

```
var a=5
var b=10
//判断
if(a>b){
    println(a)
}else{
    println(b)
}
```

此时运行程序，会看到如下的结果：

```
10
```

(3) else...if 语句

else...if 语句的嵌套语法形式如下：

```
if(表达式 1)
    语句 1
else if(表达式 2)
    语句 2
```

```
else if(表达式 3)
    语句 3
...
else if(表达式 m)
    语句 m
else
    语句 n
```

其中，当表达式 1 成立时，执行语句 1；不成立时，就判断表达式 2。当表达式 2 成立，就执行语句 2；不成立，就判断表达式 3，依此类推。它的执行流程图如图 3.8 所示。

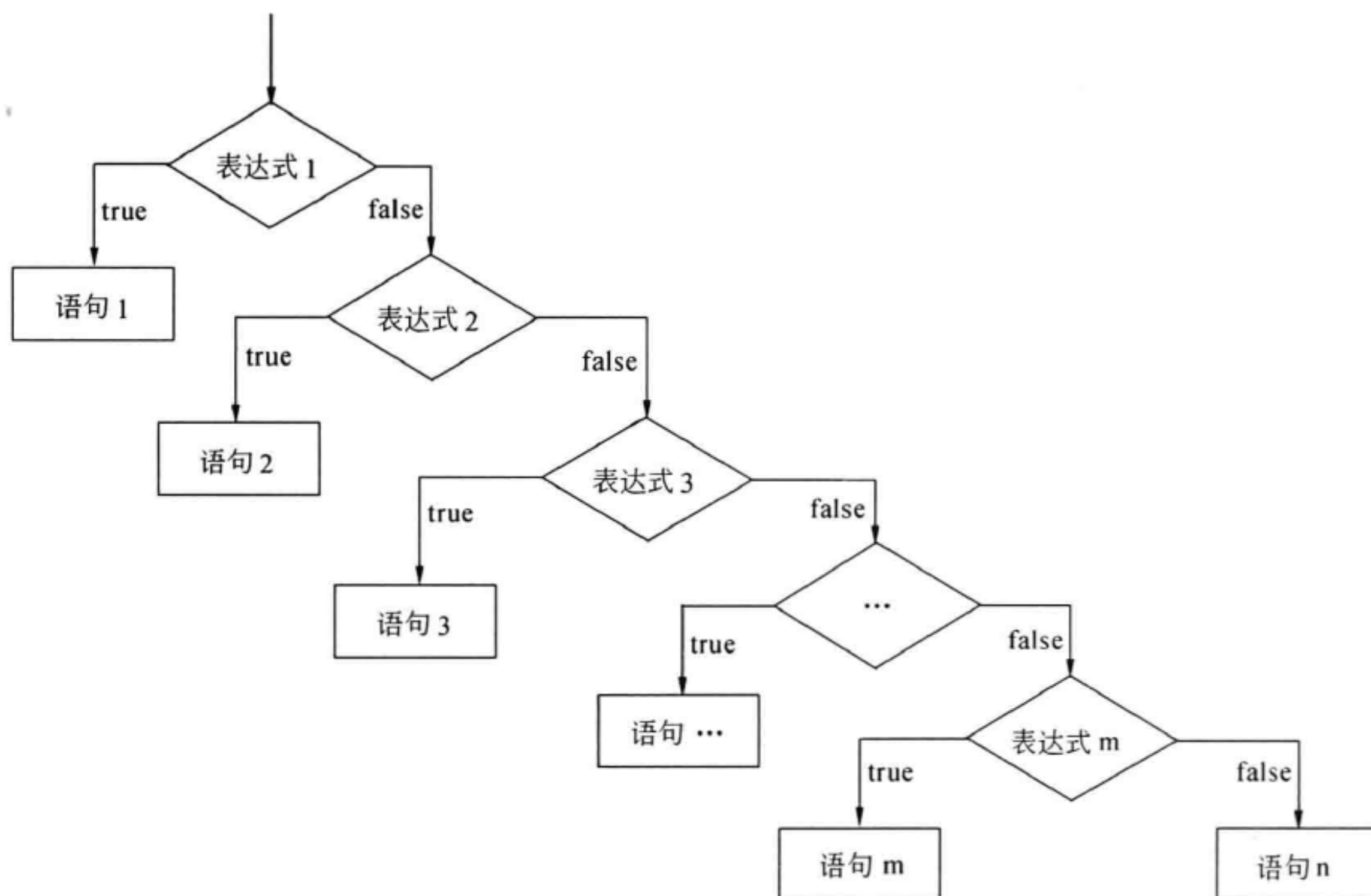


图 3.8 流程图

【示例 3-47】 以下将使用 `else...if` 语句的嵌套形式，将变量 `a` 的值输出。代码如下：

```
var a=3
if(a==0){
    println("0")
}
else if (a==1){
    println("1")
}
else if (a==2){
    println("2")
}
else if (a==3){
    println("3")
}
else{
    println("4")
}
```

此时运行程序，会看到如下的效果：

3

3. switch语句

switch 语句也是多分支语句，它的语法形式如下：

```
switch(控制表达式)
{
    case 常量或常量表达式 1:
        语句 1
    case 常量或常量表达式 2:
        语句 2
    ...
    default:
        语句 n
}
```

其中，先计算表达式的值，并逐个与其后的常量或常量表达式的值相比较，当 switch 上的表达式的值与某个 case 下的常量表达式的值相等时，即执行其后的语句。它的执行流程图如图 3.9 所示。

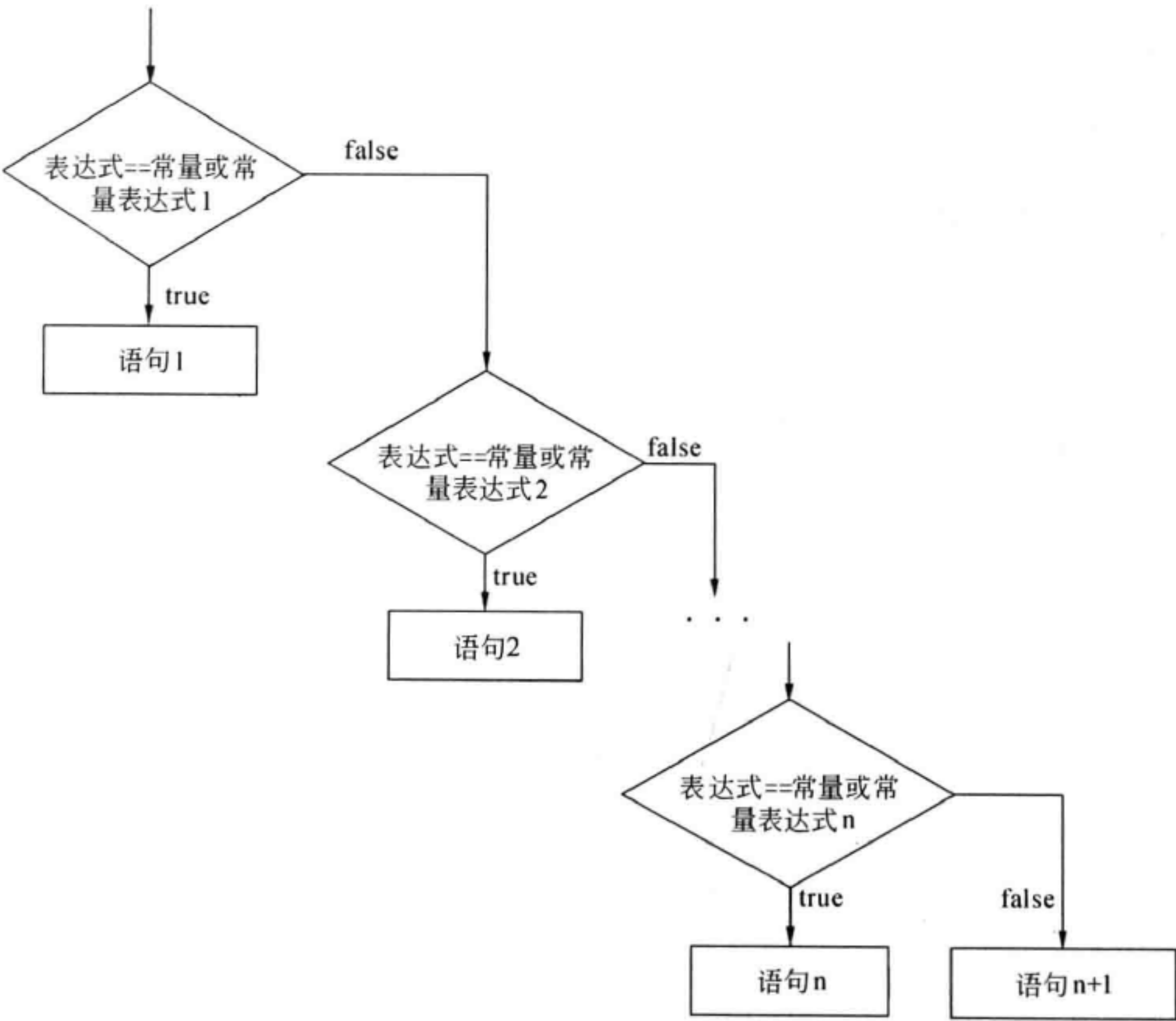


图 3.9 流程图

【示例 3-48】 以下将使用 switch 语句判断字符 "C" 在哪一个分数段，并将此分数段的分数输出。代码如下：

```
let grade="C"
```



```
switch(grade) {  
    case "A":  
        println("90~100")  
    case "B":  
        println("80~90")  
    case "C":  
        println("70~80")  
    case "D":  
        println("60~70")  
    default:  
        println("60 分以下")  
}
```

此时运行程序，会看到如下的结果：

```
70~80
```

3.8.3 循环结构

循环结构是用来在指定的条件下多次重复执行同一组语句。使用循环结构可以减小代码的编写量和时间。在 Swift 中提供了 3 种用于循环的语句：while 语句、for 语句和 do...while 语句。以下就是这 3 种语句的详细介绍。

1. while 语句

while 语句是最简单的循环语句。它的语法形式如下：

```
while(表达式)  
    语句
```

其中，表达式就是循环条件。在 while 循环语句执行时，首先要进行条件的判断，当条件成立时就执行语句，当条件不成立了，就跳出循环，其执行流程如图 3.10 所示。

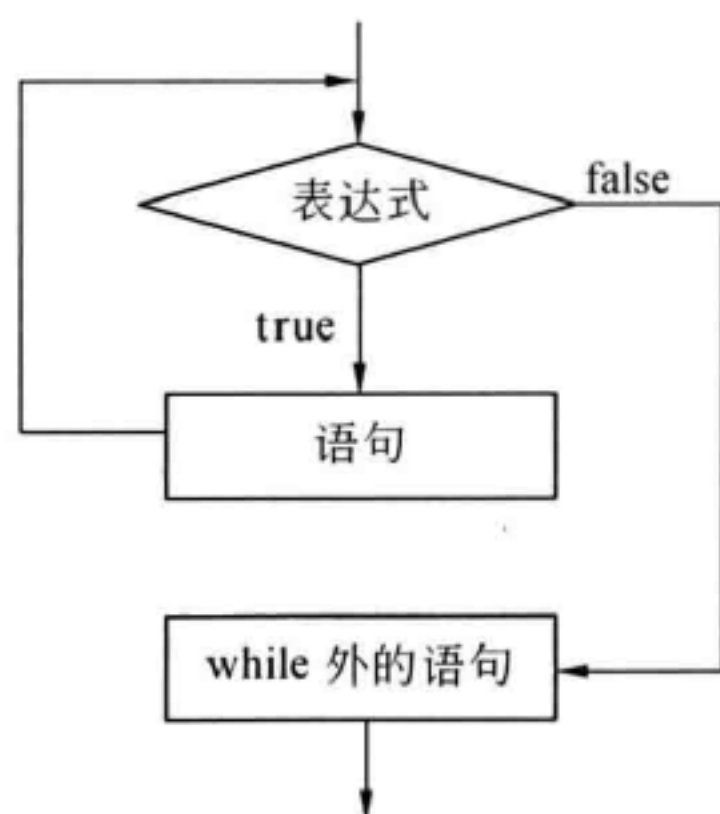


图 3.10 while 循环的执行流图

【示例 3-49】 以下就实现 while 语句实现数字 0~5 的输出。代码如下：

```
var i=0  
//循环  
while(i<=5) {  
    println(i)
```

```
i++
}
```

此时运行程序，会看到如下的结果：

```
0
1
2
3
4
5
```

2. for语句

在 Swift 中 for 语句的使用是非常灵活的。它既可用于循环次数确定的情况，也可用于循环次数不确定而只给出循环结束条件的情况。for 语句的使用方式有两种：一种是 for...in 循环，另一种是 for-condition-increment 条件循环。以下将详细讲解这两种 for 循环方式。

(1) for...in 循环

for...in 循环常常用于集合、字符串，以及数字范围的访问中。它会对数字范围、字符串、集合中的每一个元素都执行一次。其表示形式如下：

```
for 常量 in 循环的项目
    语句
```

其中，循环的项目可以是数字范围、字符串以及集合；语句也可以是一条语句，也可以是语句块。

【示例 3-50】 以下将使用 for...in 循环语句实现字符串"Swift"的遍历，即将字符串中的每一个元素都输出。代码如下：

```
var str="Swift"
//遍历，输出字符串中的字符
for index in str {
    println(index)
}
```

此时运行程序，会看到如下的结果：

```
S
w
i
f
t
```

(2) for-condition-increment 条件循环

for-condition-increment 条件循环包括了初始条件、条件语句和增量语句。它的语法形式如下：

```
for (表达式 1, 表达式 2, 表达式 3)
    语句
```

其中，表达式 1 表示对循环控制变量进行的初始化；表达式 2 表示循环的条件；表达式 3 表示对循环控制的增量。for-condition-increment 条件循环的执行分为了以下几步：

第一步，进入循环时，初始化语句，即表达式 1 首先被执行，设定好循环的常量或者

变量。

第二步，判断条件语句，即表达式 2，检查是否满足循环的条件，当条件语句为 true 时，会继续执行循环体内的语句，如果为 false 时，循环结束。

第三步，在所有循环体语句执行完毕后，增量语句，即表达式 3 被执行，然后再返回到第二步执行。它的执行流程图如图 3.11 所示。

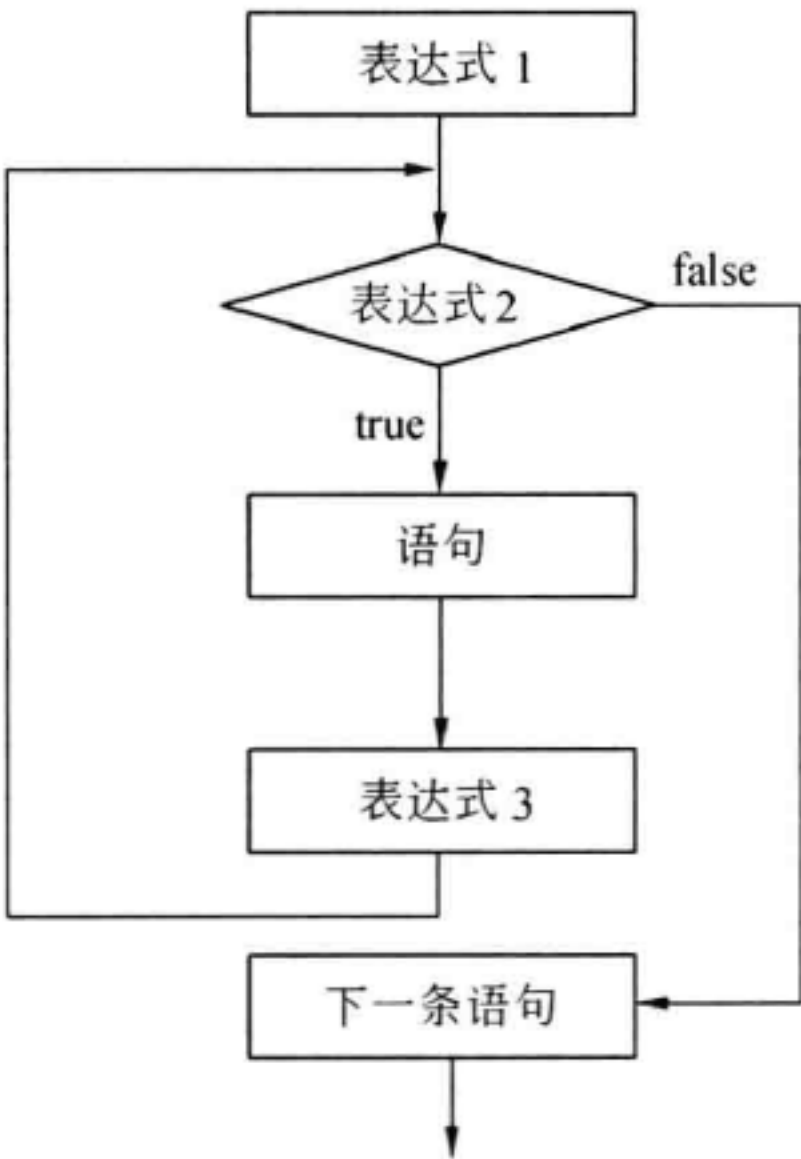


图 3.11 流程图

【示例 3-51】 以下将实现 for-condition-increment 条件循环实现 1+2+3+...+100 的值，并输出。代码如下：

```
var value:Int
var sum=0
//循环，求和
for (value=1;value<=100;++value) {
    sum+=value
}
println("sum=\(sum)")
```

此时运行程序，会看到如下的结果：

```
sum=5050
```

3. do...while语句

do...while 语句是 while 语句的一种变化形式。在 do...while 循环中，循环体中的语句会先被执行一次，然后才开始检测循环条件是否满足。它的语法形式如下：

```
do
    语句
while (条件表达式)
```

其中，当 do...while 语句开始执行时，先执行一遍 do 下面的语句，再进行 while 中的条件判断。当条件为真时，再执行 do 后面的语句；当条件为假时，就跳出 do...while 循环。它的执行流程图如图 3.12 所示。

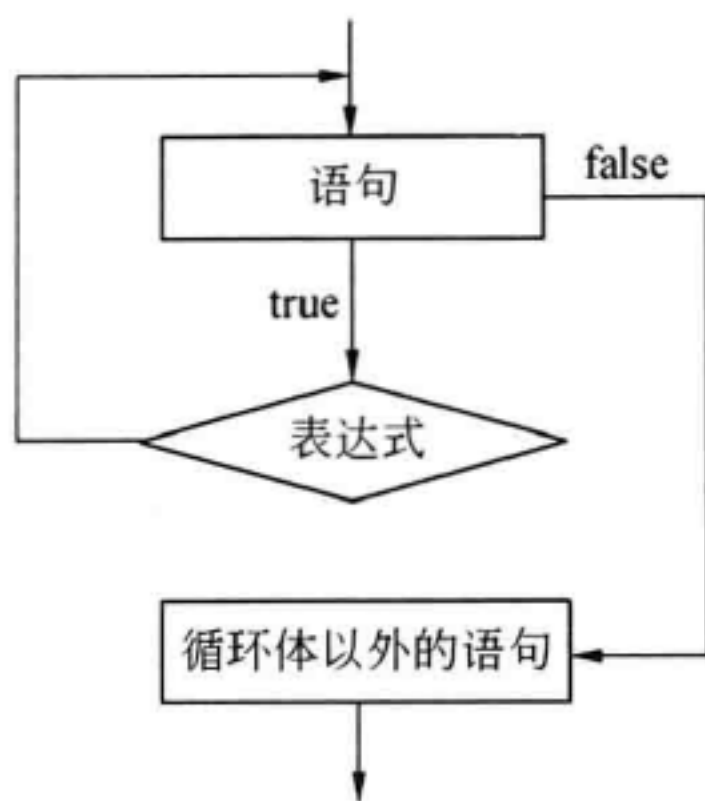


图 3.12 流程图

⚠注意：do…while 和 while 语句不同的地方是，while 语句的循环体有可能一次也不执行，但 do…while 语句的循环体至少执行一次。

【示例 3-52】 以下将数字 10086 倒序输出。代码如下：

```
var a=10086
var b:Int
//循环
do {
    b=a%10
    println(b)
    a/=10
} while (a != 0)
```

此时运行程序，会看到如下的结果：

```
6
8
0
0
1
```

3.8.4 跳转语句

跳转语句一般使用在需要提前跳出循环，或者某种条件下不执行循环而执行下一次新的一轮循环时。在 Swift 中支持 4 种跳转语句：continue、break、fallthrough 和 return。以下就是对这 4 种跳转语句的详细介绍。

1. continue 语句

如果使用了 continue 语句，一旦执行该语句，程序就会结束本次循环而执行循环体的下一次循环。continue 语句只用在 for、while、do…while 等循环体中，常与 if 条件语句一起使用。

【示例 3-53】 下面就使用 continue 语句输出 0~10 之间的偶数。代码如下：

```
var i:Int
for i=0;i<=10;++i{
    //判断是否为偶数
```

```

    if(i%2 != 0){
        continue           //使用 continue 语句结束本次循环，进入下一次循环
    }
    println(i)
}

```

此时运行程序，会看到如下的结果：

```

0
2
4
6
8
10

```

2. break

break 语句不仅可以用在 **for**、**while**、**do...while** 循环语句中，而且还可以用于我们常见的 **switch** 语句中（在 **switch** 中效果不是很明显）。如果使用了 **break** 语句，一旦执行该语句，将终止整个循环的执行。

【示例 3-54】 以下将使用 **break** 跳出循环，实现输出 0~10 之间小于 6 的数字。代码如下：

```

var i:Int
//循环
for(i=0;i<11;i++){
    //判断 i 是否大于等于 6
    if(i>=6){
        break           //使用 break 语句结束循环
    }
    println(i)
}

```

此时运行程序，会看到如下的结果：

```

0
1
2
3
4
5

```

3. fallthrough语句

fallthrough 使用在 **switch** 代码块中，实现依次执行每个 **case** 语句。

【示例 3-55】 以下将在 **switch** 中使用 **fallthrough**，让其输出 Swift 这几个字符。代码如下：

```

let str="s"
switch (str){
    case "s":
        println("S")
        fallthrough
    case "w":
        println("w")
        fallthrough
    case "i":

```

```

        println("i")
        fallthrough
    case "f":
        println("f")
        fallthrough
    default:
        println("t")
}

```

此时运行程序，会看到如下的结果：

```

S
w
i
f
t

```

📌 注意：fallthrough 只可以放在 case 后面，不可以放到 default 后。

4. return 语句

return 语句用于从一个程序中返回。换句话说，return 语句是程序控制返回到调用它的程序中，返回时可附带一个返回值。return 语句也可以用来提早结束程序的执行。

【示例 3-56】 以下将使用 return 跳出循环，实现输出 0~10 之间小于 6 的数字。代码如下：

```

var i:Int
//循环
for(i=0;i<11;i++){
    //判断 i 是否大于等于 6
    if(i>=6){
        return //使用 return 语句结束循环
    }
    println(i)
}

```

此时运行程序，会看到如下的结果：

```

0
1
2
3
4
5

```

3.8.5 标签语句

标签语句可以使用标签来标记一个循环体或者 switch 代码块。当使用 break 或者 continue 语句时，带上这个标签，就可以控制跳转该标签代表的循环或 switch 代码块了。标签语句一般放在循环或 switch 语句的行首，并且使用冒号分隔开。下面为 while 循环语句做一个标签，其语法形式如下：

标签名称: while 表达式 {


```
    语句
}
```

同样的方式适用于其他循环体和 `switch` 代码块。标签语句一般使用在 `break` 或者 `continue` 语句后面，这样就可以控制跳转该标签代表的循环或 `switch` 代码块了。

【示例 3-57】 以下将在 `break` 的后面使用标签语句。代码如下：

```
var a=0
var i=1
loop:while a<=100{
    println("开始循环第\ (i) 次")

    switch (a) {
        case 0...60:
            println("E")
        case 61...70:
            println("D")
        case 71...80:
            println("C")
            break loop
        case 81...90:
            println("B")
        default:
            println("A")
    }
    a+=10
    i++
    println("第\ (i) 次循环结束")
}
```

此时运行程序，会看到如下的结果：

```
开始循环第 1 次
E
第 2 次循环结束
开始循环第 2 次
E
第 3 次循环结束
.....
开始循环第 8 次
D
第 9 次循环结束
开始循环第 9 次
C
```

3.9 函 数

在编程中，随着处理的问题越来越复杂，代码量飞速增加。其中，大量的代码往往相互重复或者近似重复。如果不采用有效方式加以解决，代码将很难维护。为了解决这个问题，人们提出了函数这一概念。使用函数可以将特定功能的代码封装，然后在很多地方进行使用。这样既可以减少代码的编写量以及时间，又可以使用结构鲜明，便于理解。本节将讲解有关函数的内容。

3.9.1 函数的介绍

函数就是将有特定功能的语句组合在一起的形式。从不同的角度可以将函数分为不同的种类，如表 3-14 所示。

表 3-14 函数的分类	
角 度	分 类
从用户使用角度可分为	标准库函数
	用户自定义库函数
从函数的参数角度可分为	无参函数
	有参函数

一个完整的函数由 `func` 关键字、函数名、参数表，以及函数的返回值类型组成。它的语法形式如图 3.13 所示。

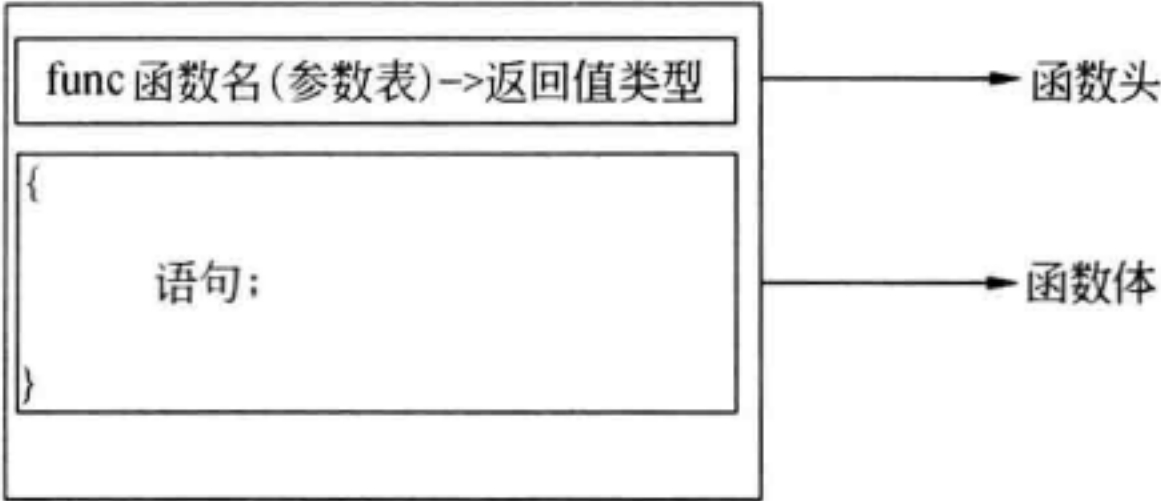


图 3.13 函数的形式

3.9.2 无参函数的使用

无参函数是指没有参数列表的函数。要想使用无参函数首先要对函数进行声明和定义。它的声明和定义是在一起进行的。声明定义的一般形式如下：

```
func 函数名 () ->返回值类型{  
    语句  
}
```

其中，语句可以是一条语句，也可以是由多条语句组合的语句块；函数名同样是一个标识符，用来表示函数要实现的功能；`->返回值类型`表示函数可能返回的值的类型（对于函数的返回值，会在后面的小节中讲解）。它是可以被省略的。如果省略，表示函数无返回值（为了便于开发者的理解，以下所讲的内容都以函数无返回值来进行讲解）。函数中的语句可以有，也可以没有，如果函数定义中没有语句，就是空语句，包含空语句的函数是最简单的函数。函数定义好以后，就可以使用该函数了，这里称之为函数的调用。函数调用的形式如下：


```
函数名 ()
```

【示例 3-58】 以下将定义一个函数名为 `printSwift()`的函数，此函数用来输出字符串 "Swift"。代码如下：

```
//声明定义函数
func printSwift(){
    println("Swift")
}
printSwift() //调用函数
```

此时运行程序，会看到如下的结果：

```
Swift
```

 **注意：**函数一般都是先声明和定义，后调用的。

3.9.3 有参函数的使用

所谓有参函数，也就是函数头的参数表不为空。要想使用有参函数首先要对函数进行声明和定义。它的声明和定义也是在一起进行的。声明定义的一般形式如下：

```
func 函数名(参数名 1:数据类型, 参数名 2:数据类型, ……) ->返回值类型{
    语句
}
```

其中，无参函数和有参函数的区别在于有无参数列表。参数列表由参数名和数据类型组成，其中使用 **:** 冒号将它们分隔开。参数列表中的参数可以有多个。定义时使用参数是为了接收调用时传递的数据。有参函数调用的语法形式如下：


```
函数名(参数值 1, 参数值 2, …)
```

【示例 3-59】 以下将实现问候某一个人的功能。代码如下：

```
//有参函数的声明定义
func hello(Name: String){
    println("你好,\(Name)")
}
let personName1="杨过"
hello(personName1) //调用
let personName2="小龙女"
hello(personName2)
```

此时运行程序，会看到如下的结果：

```
你好,杨过
你好,小龙女
```

 **注意：**在有参函数中，参数可以是一个，也可以是多个。

【示例 3-60】 以下将输出某一范围内包含的数字。代码如下：

```
//函数声明定义
func range(start:Int,end:Int){
    var i=start
    var j=end
    //循环
    for i;i<=j;++i{
```



```
        println(i)
    }
}
range(0,5) //调用
```

此时运行程序，会看到如下的结果：

```
0
1
2
3
4
5
```

3.9.4 函数的参数的注意事项

在使用有参函数时，需要注意参数的一些事项。

1. 本地参数名

本地参数名就是定义函数名时，在参数列表中所写的参数名，它只可以在函数主体内使用。如以下的一个代码片段，它定义了一个函数名为 `fun` 的函数，在此函数的参数列表中定义的就是本地参数名。

```
func fun(start:Int,end:Int,str:String) {
    .....
}
```

在以上的代码中，定义的参数名 `start`、`end`、`str` 都是本地参数名。它只可以在函数本身的代码中使用，在调用时就不可以使用了。

2. 外部参数名

外部参数名是为了让函数中的参数明确，以及便于理解。外部参数名需要写在本地参数名之前，并使用空格将其分开。它的一般形式如下：

```
func 函数名(外部参数名 本地参数名:数据类型) ->返回值类型 {
    .....
}
```

对于外部参数名的函数，调用形式如下：


```
函数名(外部参数名:参数值)
```

【示例 3-61】 以下将在输出字符串的函数中声明外部参数名。代码如下：

```
func printstr(string str:String) {
    println(str)
}
printstr(string: "Hello") //调用函数
```

此时运行程序，会看到如下的结果：

```
Hello
```

 **注意：**在本地参数名前面添加一个#，可以将本地参数名变为外部参数名，其语法形式如下：

```
func 函数名(#本地参数名:数据类型) ->返回值类型{
    .....
}
```

它的调用形式如下：

函数名(外部参数名:参数值)

【示例 3-62】 以下将实现计算两个整数的和的功能。代码如下：

```
func add(#value1:Int,#value2:Int){
    var sum=value1+value2
    println("\(value1)+\(value2)=\(sum)")
}
add(value1: 60, value2: 6) //求和
//函数调用
```

此时运行程序，会看到如下的结果：

```
60+6=66
```

3. 为参数设置默认值

在定义函数的时候，可以为参数设定默认值。这样，在调用的时候，就不用再传递该参数的值。以下我们将分别为本地参数和外部参数设置默认值。

(1) 为本地参数设置默认值

为本地参数设置默认值的语法形式如下：

```
func 函数名(参数名 1:数据类型=值,参数名 2:数据类型=值,……) ->返回值类型{
    语句
}
```

【示例 3-63】 以下将在输出指定范围的函数中为参数设置默认值。代码如下：

```
func printInt(start:Int=0,end:Int=5){
    var i=start
    //循环，输出值
    for i;i<=end;++i{
        println(i)
    }
}
printInt() //为参数设定的默认值
```

此时运行程序，会看到如下的结果：

```
0
1
2
3
4
5
```

(2) 为外部参数设置默认值

为外部参数设置默认值，其语法形式如下：


```
func 函数名(外部参数名 1 参数名 1:数据类型=值, 外部参数名 2 参数名 2:数据类型=
值, ...) ->返回值类型{
    语句
}
```

【示例 3-64】 以下将实现两个整数的减法运算。代码如下：

```
func subtraction(minuend a:Int=90, subtrahend b:Int=24){
    //为参数设定的默认值
    var c=a-b
    println("两数相减最后结果为: \(c)")
}
subtraction()
```

此时运行程序，会看到如下的结果：

两数相减最后结果为：66

4. 可变参数

使用可变参数，可以使一个参数接受零个或多个指定类型的值。设定一个可变参数需要在参数类型名后添加...

【示例 3-65】 以下将实现求几个数的平均值的功能。代码如下：

```
func average(numbers: Double...){
    //可变参数
    var total: Double = 0
    //求和
    for number in numbers {
        total += number
    }
    println(total/Double(numbers.count))
}
average(1, 2, 3, 4, 5)
average(20,50,70)
```

//函数调用
//函数调用

此时运行程序，会看到如下的结果：

3.0
46.66666666666667

5. 常量参数和变量参数

在函数中，参数的默认都是常量，常量的值是不可以改变的，如果想要改变参数中的值，需要将常量参数改变为变量参数。变量参数的定义就是在参数名前使用一个 var 关键字。

【示例 3-66】 以下将实现求一个数的中间值。代码如下：

```
func midvalue(var value:Int){
    //变量参数
    value=value/2
    println("中间数为: \(value)")
}
midvalue(50)
```

此时运行程序，会看到如下的结果：

中间数为：25

6. 输入-输出参数

以上函数中所使用的参数只可以在函数内部发生改变。如果开发者想用一个函数来修改参数的值，并且想让这些变化在函数调用后仍然有效，这时，需要定义输入-输出参数。它的定义是通过在参数名前加入 `inout` 关键字实现的。其语法形式如下：

```
func 函数名(inout 参数名: 数据类型, ...) {
    ...
}
```

输入-输出参数都有一个传递给函数的值，将函数修改后，再从函数返回来替换原来的值。其调用形式如下：

```
函数名(&参数, ...)
```

其中，参数前面加上 `&` 运算符。

【示例 3-67】 以下将实现数的交换。代码如下：

```
func swapTwoInt(inout number1: Int, inout number2: Int) { //输入-输出参数
    //实现两个整数的交换
    let temp = number1
    number1 = number2
    number2 = temp
}
var a=6
var b=8
println("交换前")
println("a=\(a)")
println("b=\(b)")
println("交换后")
swapTwoInt(&a, &b) //调用
println("a=\(a)")
println("b=\(b)")
```

此时运行程序，会看到如下的结果：

```
交换前
a=6
b=8
交换后
a=8
b=6
```

3.9.5 函数的返回值

在以上的示例中我们都没有使用到返回值。在函数中可以有返回值，也可以没有返回值。函数是否有返回值，以及返回值的数据类型都是与函数声明定义有关的。本小节将主要讲解有返回值的情况。

1. 具有一个返回值的函数


在一个函数中，返回一个值是最常见到的，也是最为简单的。开发者希望在函数中返

回某一数据类型的值，必须要在函数声明和定义时为函数设定一个返回的数据类型，并使用 `return` 语句进行返回。其中，`return` 语句的一般表示形式如下：

```
return 表达式
```

其中，表达式可以是符合 Swift 标准的任意表达式。而具有返回值的函数声明和定义形式如下：

```
func 函数名(参数列表) -> 返回值类型 {
    语句
    return 表达式
}
```

 **注意：** 返回的表达式类型必须和函数的返回值类型一致。

【示例 3-68】 以下将实现两个数的乘法运算。代码如下：

```
func multiplication(value1:Int,value2:Int)->Int{
    return value1*value2
}
println(multiplication(10, 6))
```

此时运行程序，会看到如下所有的结果：

```
60
```

2. 具有多个返回值的函数

函数不仅可以返回一个返回值，还可以返回多个，这时就需要使用到元组类型。其语法形式如下：

```
func 函数名(参数列表) -> (数据类型 1, 数据类型 2, 数据类型 3, ...) {
    .....
    return (表达式 1, 表达式 2, 表达式 3)
}
```

【示例 3-69】 以下将计算两个整数和两个字符串的最大值，并返回。代码如下：


```
func
maxvalue(value1:Int,value2:Int,value3:String,value4:String)->(Int,String) {
    var maxInt=0
    var maxString:String
    //判断两个整数哪个最大
    if(value1<value2){
        maxInt=value2
    }else{
        maxInt=value1
    }
    //判断两个字符串哪个最大
    if(value3<value4){
        maxString=value4
    }else{
        maxString=value3
    }
}
```



```
    return (maxInt,maxString) //返回
}
println(maxvalue(5, 1, "Hello", "Swift"))
```

此时运行程序，会看到如下的结果：

```
(5, Swift)
```

 **注意：**在->后面的返回值类型的个数要和 return 中的返回值个数一致，否则程序就会出现错误。

3.9.6 函数类型

在 Swift 中，每一个函数都是一个特定的类型，这种类型被称为函数类型。它由参数类型和返回值类型构成。例如，以下代码就是一个具有参数类型和返回值类型的函数：

```
func add(a: Int, b: Int) -> Int {
    return a + b
}
```

这个函数的类型就是(Int,Int)->Int，开发者可以理解为函数类型有两个 Int 整型参数，并返回一个 Int 整型值。在 Swift 中，除了具有参数列表和返回值类型的函数外，还有不带参数和返回值类型的函数，如以下代码是一个不带参数和返回值的函数：

```
func printHelloWorld() {
    println("Hello,World")
}
```

函数 printHelloWorld()的类型是()->()。由于函数没有参数，返回 void，所以该类型在 Swift 中相当于一个空元组，也可以简化为()。函数类型可以不让值作为一种类型进行使用，开发者可以像使用任何其他类型一样使用函数类型。其语法形式如下：

```
let/var 常量名/变量名:函数类型=函数名
```

或者：


```
let/var 常量名/变量名=函数名
```

【示例 3-70】 以下将使用一个变量引用函数。代码如下：

```
func add(value1:Int,value2:Int)->Int{
    return value1+value2
}
var function:(Int,Int)->Int=add
println(function(10,10))
```

此时运行程序，会看到如下的结果：

```
20
```

 **注意：**由于 Swift 具有自动推断类型的能力，所以可以在声明变量后直接赋值，不需要单个为变量去声明类型，所以以上为变量赋值的代码可以改写为：


```
var function = add
```

一般函数类型可以作为参数和返回值类型进行使用。以下将是对这两种使用方法的详细讲解。

1. 使用函数类型作为参数


开发者可以使用函数类型作为另一个函数的参数类型。

【示例 3-71】 以下将函数类型作为函数 `printresult` 的一个参数，实现输出结果。代码如下：

```
//两数相加
func add(a: Int, b: Int) -> Int {
    return a + b
}
//两数相乘
func multiply(a: Int, b: Int) -> Int {
    return a * b
}
//输出结果
func printresult(fun: (Int, Int) -> Int, a: Int, b: Int) {
    println(fun(a, b))
}
printresult(add, 3, 2)
printresult(multiply, 3, 2)
```

此时运行程序，会看到如下的结果：

```
5
6
```

 **注意：**在此代码中定义了3个函数。第3个函数 `printresult` 有3个参数：第一个参数为 `fun`，类型为 `(Int, Int) -> Int`，开发者可以传入任何这种类型的函数；第二个参数和第三个参数分别为 `a` 和 `b`，它们的类型都是 `Int` 型，这两个值是函数的输入值。当第一次调用 `printresult` 函数时，它传入了 `add` 函数和 3、5 两个整数。这时，它又会调用函数 `add`，将 3 和 5 作为函数 `add` 的输入值，并输出结果。第二次调用也类似，`printresult` 会调用 `multiply` 函数。

2. 使用函数类型作为返回值类型

开发者还可以将函数类型作为返回值类型来使用。此时需要在 `->` 后面写入一个完整的函数类型，其语法形式如下：

```
func 函数名(参数列表) -> 函数类型 {
    ...
}
```

【示例 3-72】 以下将通过给定的值，输出一系列的值。代码如下：

```
//返回一个比输入值大 1 的值
```

```
func stepForward(input: Int) -> Int {
    return input + 1
}
//返回一个比输入值小 1 得值
func stepBackward(input: Int) -> Int {
    return input - 1
}
//选择返回哪个函数
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    return backwards ? stepBackward : stepForward
}
var currentValue = 5
let moveNearerToZero = chooseStepFunction(currentValue<0)
while currentValue < 11 {
    println("\(currentValue)")
    currentValue = moveNearerToZero(currentValue)
}
```

此时运行程序，会看到如下的结果：

```
5
6
7
8
9
10
```

3.9.7 常用的标准函数

以上所讲的函数都是用户自定义的函数。除了自定义函数外，Swift 还提供了标准函数，这些标准函数共有 74 个。表 3-15 是这 74 个标准函数中的常用函数。

表 3-15 常用的标准函数

函 数	功 能
abs()	求数值的绝对值
max()	获取几个参数的最大值
min()	获取几个参数的最小值
maxElement()	获取一个序列中最大值的元素
minElement()	获取一个序列中最小值的元素
contains()	判断一个序列中是否包含指定的元素
sort()	排序
reverse()	将序列中元素的倒序排列

【示例 3-73】 以下将使用表 3-15 中提供的函数实现求最大值、最小值的元素、排序、倒序的功能。代码如下：

```
let maxvalue=max(2, 1, 1000, 100,500) //获取最大值
println(maxvalue)
```

```
var array=[10,1,6,7,8,]  
let minvalue=minElement(array)           //获取数组中的最小值的元素  
println(minvalue)  
sort(&array)                               //排序  
println(array)  
var arr=[2,8,9,5]  
println(reverse(arr))                     //倒序
```

此时运行程序，会看到如下的结果：

```
1000                                     //最大值  
1                                       //最小值  
[1, 6, 7, 8, 10]                       //排序  
[5, 9, 8, 2]                           //倒序
```

3.9.8 函数的嵌套

在 Swift 中，函数还可以调用其他函数，从而形成嵌套调用。嵌套调用的形式往往有两种：一种是在一个函数中调用其他函数；另一种是在一个函数中调用自身函数。以下将对这两种调用形式进行详细讲解。

1. 嵌套调用

所谓函数的嵌套，也就是在函数定义时，调用了一个或多个其他的函数。函数嵌套调用的形式如图 3.14 所示。

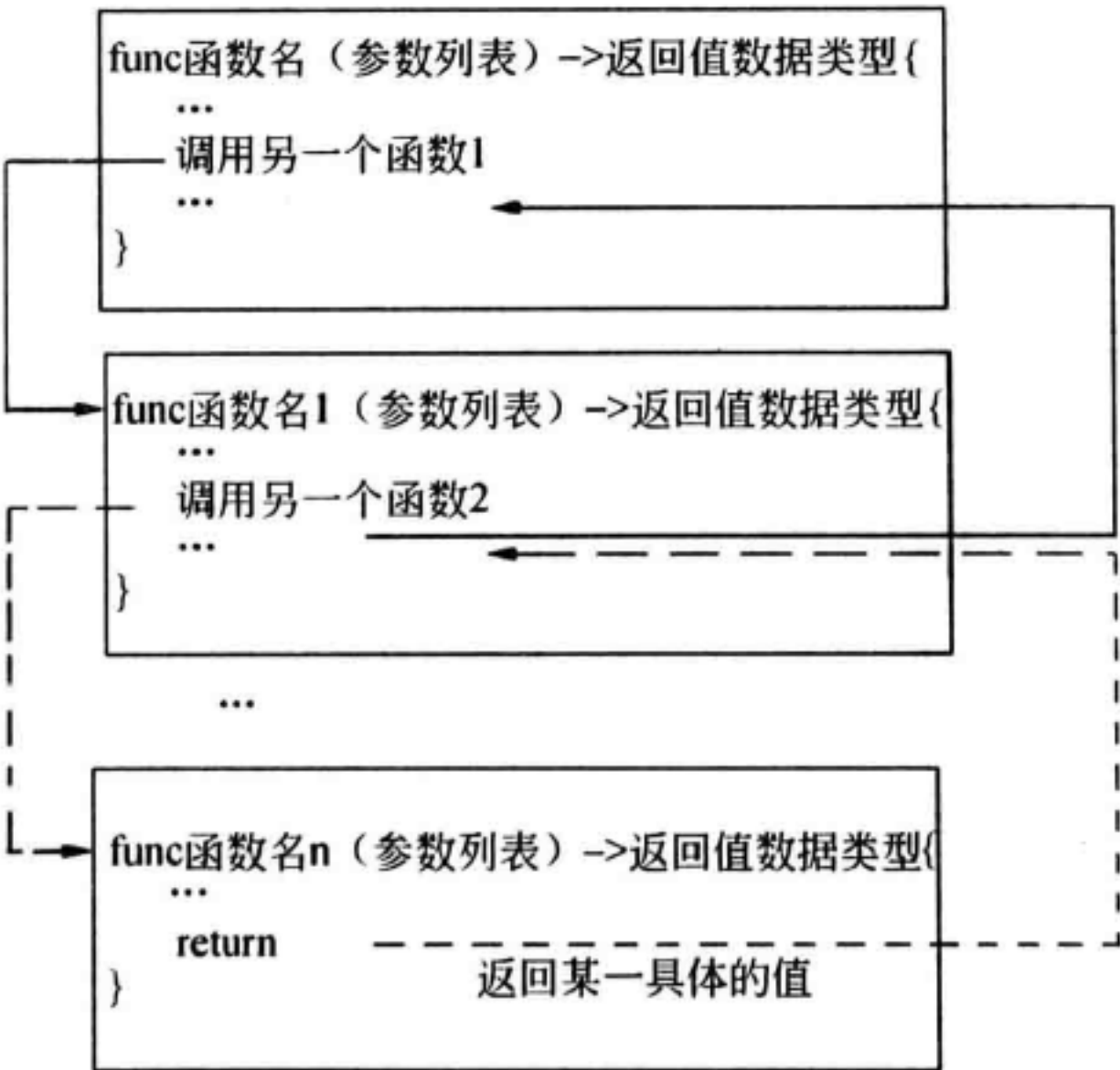


图 3.14 嵌套调用

【示例 3-74】 以下通过使用函数的嵌套，实现求 $s=1^2!+2^2!$ 。代码如下：

```
func f1(q:Int)->Int{  
    var c:Int=1
```



```
var i:Int
for i=1;i<=q;++i{
    c=c*i
}
return c;                                     //获取阶乘的值
}
func f2(p:Int)->Int{
    var k:Int
    var r:Int
    k=p*p                                     //求平方
    r=f1(k)                                  //调用函数 f2 ()，计算阶乘
    return r                                //获取平方后阶乘的值
}
//求阶乘
var i:Int
var s:Int=0
for i=2;i<=3;i++ {
    s=s+f2(i);
}
println("s=\(s)")
```

此时运行程序，会看到如下的结果：

s=362904

2. 递归调用

递归调用是嵌套调用的一种特殊情况，它也可以被称为递归调用。它在调用函数的过程中调用了该函数本身。递归调用的形式如图 3.15 所示。

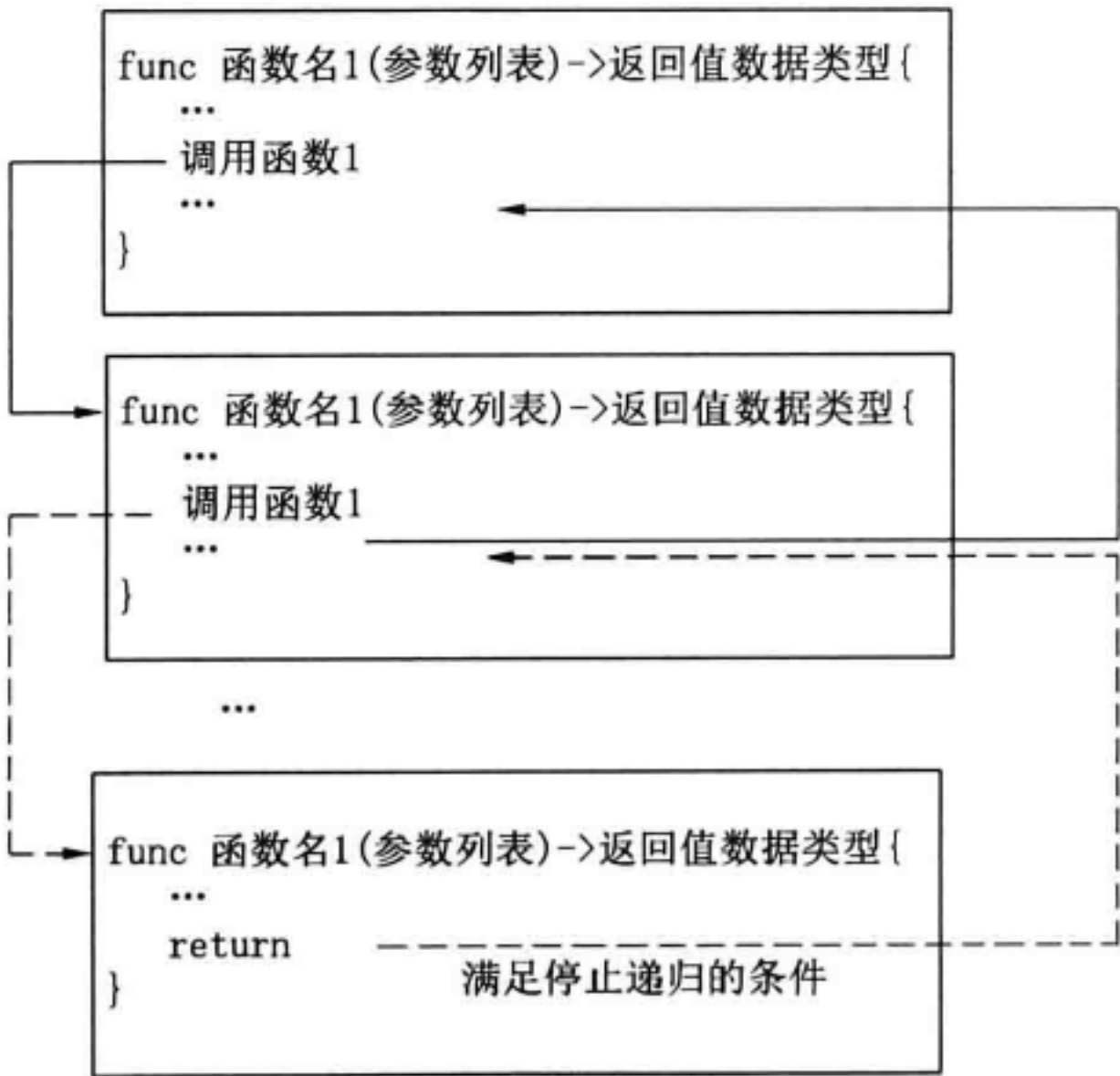


图 3.15 递归调用

【示例 3-75】 以下程序通过使用函数的递归调用，求 sum=i!+2!+3!+4!+5!。代码如下：

```

func factorial(var value:Int)->Int{ //求阶乘
    if(value==1){
        return value
    }else{
        return value * factorial(value-1)
    }
}
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view, typically from a nib.
    var i:Int
    var sum=0
    for(i=1;i<=5;i++){
        sum=sum+factorial(i)
    }
    println(sum)
}

```

此时运行程序，会看到如下的结果：

153

3.10 闭包

闭包是自包含的函数代码块，可以在代码中被传递和使用。在 3.9 节中所讲的函数其实就是特殊的闭包。本节将主要讲解闭包表达式、使用闭包表达式的注意事项、Trailing 闭包，以及捕获值等内容。

3.10.1 闭包表达式

闭包表达式是一种利用简洁语法构建内联（内联类似于 C 语言中的宏定义）闭包的方式。闭包表达式（闭包函数）的语法形式如下：

```

{ (参数列表) -> 返回值类型 in
    语句
}

```

其中，参数可以是常量、变量和输入-输出参数，但没有默认值。开发者也可以在参数列表的最后使用可变参数，而元组也可以作为参数和返回值。关键字 `in` 表示闭包的参数和返回值类型定义已经完成，闭包函数体即将开始。闭包表达式和 3.9 节中提到的函数一样，可以分为无参闭包表达式和有参闭包表达式。

1. 无参的闭包表达式

无参形式的闭包表达式语法形式如下：

```
{ ()->返回值类型 in
    语句
}
```

它定义的语法形式如下：

let/var 闭包表达式常量名称/闭包表达式变量名称/=无参形式的闭包表达式

它调用的语法形式如下：

闭包表达式常量名称/闭包表达式变量名称()

【示例 3-76】 以下将输出字符串"Swift"。代码如下：

```
var printStr={ () in
    println("Swift")
}
printStr()
```

此时运行程序，会看到如下的结果：

Swift

2. 有参的闭包表达式

具有参数的闭包表达式的语法形式如下：

```
{ (参数名 1:数据类型, 参数名 2:数据类型, ...) ->返回值类型 in
    语句
}
```

它定义的语法形式如下：

let/var 闭包表达式常量名称/闭包表达式变量名称/=具有参数的闭包表达式

它的调用形式如下：

闭包表达式常量名称/闭包表达式变量名称(参数值 1, 参数值 2...)

【示例 3-77】 以下将实现两个整数的求和运算。代码如下：

```
var add = {(s1: Int, s2: Int) -> Int in
    var sum=s1+s2
    return sum
}
println(add(10,20))
```

此时运行程序，会看到如下的结果：

30

其实，闭包表达式最常用在其他的函数中，并不是单独的去使用它。

【示例 3-78】 以下将闭包表达式作为函数的一部分，实现判断在数组中是否有大于 500 或者 40 的元素。代码如下：

```
func copare(arr:[Int],value:Int,cb:(Num:Int,Value:Int)->Bool)->Bool{
    //遍历数组
```



```

    for item in arr{
        //判断闭包是否为真
        if(cb(Num: item,Value: value)){
            return true
        }
    }
    return false
}
var array = [20,80,100,50,20]
//使用闭包判断是否在数组中有大于 500 的元素
var v1=copare(array,500,{(num:Int,value:Int)->Bool in
    return num>value
})
//判断结果并输出
if v1==true {
    println("数组 array 中有比 500 大的元素")
}else{
    println("数组 array 中没有比 500 大的元素")
}
//使用闭包判断是否在数组中有大于 40 的元素
var v2=copare(array,40,{(num:Int,value:Int)->Bool in
    return num>value
})
//判断结果并输出
if v2==true {
    println("数组 array 中有比 40 大的元素")
}else{
    println("数组 array 中没有比 40 大的元素")
}

```

此时运行程序，会看到如下的结果：

```

数组 array 中没有比 500 大的元素
数组 array 中有比 40 大的元素

```

在使用闭包表达式时需要注意以下几点（以下都是以【示例 3-78】来说明的）：

（1）推断类型

`copare()`函数的第3个参数是闭包表达式，它是类型为`(num:Int,value:Int)->Bool`的函数。

由于 Swift 可以推断其参数和返回值的类型，所以`->`和围绕在参数周围的括号可以省略，如以下的代码：

```

var v1=copare(array,500,{(num,value) in
    return num>value
})

```

（2）省略 return

单行表达式闭包可以通过隐藏 `return` 关键字来隐式返回单行表达式的结果，可以将上面的例子进行修改：

```

var v1=copare(array,500,{(num,value) in
    num>value
})

```

```
})
```

(3) 简写参数名

Swift 为内联函数提供了参数名缩写功能，开发者可以通过\$0、\$1、\$2 来顺序的调用闭包的参数。如果在闭包表达式中使用参数名称缩写，可以在闭包参数列表中省略对其的定义，并且对应参数名称缩写的类型会通过函数类型进行推断。in 关键字也同样可以被省略，因为此时闭包表达式完全由闭包函数体构成，将上面的例子进行修改如下：

```
var v1=copare(array, 500, {
    $0 > $1
})
```

(4) 写在一行

当闭包的函数体部分很短时可以将其写在同一行，如下代码：

```
var v1=copare(array, 500, {$0 > $1})
```

(5) 运算符函数


在 Swift 中 String 类型定义了关于大于号(>)的字符串实现，其作为一个函数接受两个 String 类型的参数并返回 Bool 类型的值。而这正好与以上代码 sort 函数的第二个参数需要的函数类型相符合。因此，可以简单地传递一个大于号，Swift 可以自动推断出你想使用大于号的字符串函数实现：

```
var v2=copare(array, 500, >)
```

3.10.2 Trailing 闭包

如果开发者需要将一个很长的闭包表达式作为最后一个参数传递给函数，可以使用 Trailing 闭包，它可以增强函数的可读性。Trailing 闭包的一般形式如下：

```
func someFunctionThatTakesAClosure(closure: () -> ()) {
    //函数主体部分
}
//以下不是使用 trailing 闭包进行的函数调用
someFunctionThatTakesAClosure({
    //闭包主体部分
})
//以下是使用 trailing 闭包进行的函数调用
someFunctionThatTakesAClosure() {
    //闭包主体部分
}
```

 **注意：**trailing 闭包是一个写在函数括号之后的闭包表达式，函数支持将其作为最后一个参数调用。它一般使用在当闭包很长以至于不能在同一行进行编写的代码中。

【示例 3-79】 以下将实现将数字改为英文的功能，代码如下：

```
//创建字典
let digitNames = [0: "Zero", 1: "One", 2: "Two", 3: "Three", 4: "Four", 5:
```



```

"Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"]
//创建数组
let numbers = [1,12,521,8023,12345]
//以下是使用 trailing 闭包进行的函数调用, 实现将数字转为英文
let strings = numbers.map {
    (var number) -> String in
    var output = ""
    while number > 0 {
        output = digitNames[number % 10]! + output
        number /= 10
    }
    return output
}
//遍历并输出
for index in strings{
    println(index)
}

```

此时运行程序, 会看到如下的结果:

```

One
OneTwo
FiveTwoOne
EightZeroTwoThree
OneTwoThreeFourFive

```

3.10.3 捕获值

闭包可以在其定义的上下文中捕获常量或变量。

【示例 3-80】 下面就使用 `incrementor()` 函数从上下文中对值 `runningTotal` 和 `amount` 进行捕获。代码如下:

```

func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    //定义函数 incrementor(), 实现 runningTotal 的增加
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}
//赋值
var a = makeIncrementor(forIncrement: 10)
//输出
println("输出 a 的增量")
println(a())
println(a())
println(a())
//赋值, 输出
var b = makeIncrementor(forIncrement: 5)

```



```
println("输出 b 的增量")
println(b())
println(b())
println(b())
```

此时运行程序，会看到如下的结果：

```
输出 a 的增量
10
20
30
输出 b 的增量
5
10
15
```

第4章 Swift 高级语法

在第3章中讲解了 Swift 的基础语法，这些语法在其他的编程语言中都是存在的。本章将讲解 Swift 的高级语法，如类、继承、枚举、泛型等内容。其中，部分内容是 Swift 特有的语法，需要读者仔细学习。

4.1 类

类实际是一种新的数据类型，也是实现抽象类型的工具。本节将讲解类的创建、实例化对象、定义属性，以及方法等内容。

4.1.1 创建类

在 Swift 中，类的创建类似于 C# 和 Java 等，需要使用到关键字 `class`，其语法形式如下：

```
class 类名{  
    \\具体内容  
}
```

📌 注意：类名可以使用“大骆驼拼写法”方式来命名（如 `SomeClass`），以便符合标准 Swift 类型的大写命名风格（如 `String`、`Int` 和 `Bool`）。

4.1.2 实例化对象

实例化对象也可以称为创建类的实例。它是指用类创建对象的过程，通俗点讲就是声明并创建对象。其语法形式如下：

```
var/let 对象名=类名()
```

【示例 4-1】 以下会创建一个类名为 `NewClass` 的类，然后再进行实例化。代码如下：

```
class NewClass{  
  
}  
var newclass=NewClass()
```

📌 注意：在类中如果没有任何内容就表示此类为空类。一般在编程语言中空类是很常见的，如果开发者没有想好在此类中写什么，但是此类是必不可少的时候，就先创建一个空类。

4.1.3 属性

在 Swift 中属性一般可以分为存储属性、计算属性和类型属性。以下就是对这 3 种属性的介绍。

1. 存储属性

存储属性就是存储特定类中的一个常量或者变量。根据数据是否可变，分为常量存储属性和变量存储属性，其语法形式如下：

```
let 常量存储属性名:数据类型=初始值
var 变量存储属性名:数据类型=初始值
```

其中，使用关键字 `let` 定义的存储属性为常量存储属性；使用 `var` 定义的存储属性为变量存储属性。存储属性一般都是可以进行访问的，访问存储属性的语法形式如下：

对象名.常量存储属性名/变量存储属性名

【示例 4-2】 以下将在创建的类中定义 3 个存储属性，然后分别对这 3 个属性进行访问。代码如下：

```
class NewClass{
    //定义存储属性
    let value1=50
    let value2=10.00
    let value3="Hello,Swift"
}
var newclass=NewClass()
//访问存储属性
println("value1=\(newclass.value1)")
println("value2=\(newclass.value2)")
println("value3=\(newclass.value3)")
```

在使用存储属性时，需要注意以下两点。

(1) 存储属性除了可以进行访问外，还可以进行修改。修改存储属性的一般语法形式如下：

对象名.存储属性=修改的内容

【示例 4-3】 以下将实现存储属性的修改。代码如下：

```
class NewClass{
    var secondValue:String="Hello"
}
let newclass=NewClass()
println("修改前: secondValue=\(newclass.secondValue)")
newclass.secondValue="Swift" //修改存储属性
println("修改后: secondValue=\(newclass.secondValue)")
```

此时运行程序，会看到如下的结果：

```
修改前: secondValue=Hello
修改后: secondValue=Swift
```


(2) 如果开发者只有在第一次调用存储属性时才能确定初始值,这时需要使用延迟存储属性实现。它的定义一般需要使用关键字 **lazy** 实现的,其语法形式如下:

```
lazy var 属性名:数据类型=初始内容
```

【示例 4-4】 以下将在类中定义一个延迟属性。代码如下:

```
class NewClass1 {
    var fileName = 8023
}
class NewClass2 {
    lazy var importer = NewClass1()           //延迟属性
    var data = [String]()
}
let manager = NewClass2()
manager.data += ["Some more data"]
println(manager.data)
println(manager.importer.fileName)
```

此时运行程序,会看到如下的结果:

```
[Some more data]
8023
```

在此代码中需要注意,在没有调用 `manager.importer.fileName` 时, `importer` 对象的属性还没有被创建。

2. 计算属性

计算属性不存储值,而是提供了一个 `getter` 和 `setter` 分别进行获取值和设置其他属性的值。`getter` 使用 `get` 关键字进行定义,其一般形式如下:

```
get{
    ...
    return 某一属性值
}
```

`setter` 使用 `set` 关键字进行定义,其一般语法形式如下:

```
set(参数名称){
    ...
    属性值=某一个值
    ...
}
```

在计算属性中,可以同时包含 `getter` 和 `setter`,其一般定义形式如下:

```
var 属性名:数据类型{
    get{
        ...
        return 某一属性值
    }
    set(参数名称){
        ...
        属性值=某一个值
        ...
    }
}
```

【示例 4-5】 以下将实现华氏温度和摄氏温度转换的功能。代码如下：

```
class DegreeClass{
    var degree=0.0
    //计算属性
    var cal :Double{
        get{
            var centigradedegree=(degree-32)/1.8
            return centigradedegree
        }
        set(centigradedegree){
            degree=1.8*centigradedegree+32
        }
    }
}
var degreeClass=DegreeClass()
degreeClass.cal=(10.0)
println(degreeClass.cal)
println(degreeClass.degree)
```

此时运行程序，会看到如下的结果：

```
10.0
50.0
```

在计算属性中，需要注意以下两点。

(1) 如果计算属性的 setter 没有定义表示新值的参数名时，则可以使用默认名称 newValue。

(2) 在计算属性中，可以只有一个 getter，称为只读计算属性。只读计算属性可以返回一个值，但不能设置新的值。

3. 类型属性

类型属性就是不需要对类进行实例化就可以使用的属性。它需要使用关键字 class 进行定义，其定义形式如下：

```
class var 类型属性名:数据类型{
    ...
    返回一个值
}
```

类型属性也是可以被访问的，其访问类型属性的一般形式如下：

```
类名.类型属性
```

【示例 4-6】 以下将定义一个类型属性，然后进行遍历输出。代码如下：

```
class NewClass{
    //类型属性
    class var value:String{
        return "Swift"
    }
}
//访问类型属性，并遍历
for index in NewClass.value{
    println(index)
}
```


此时运行程序，会看到如下的结果：

```
S
w
i
f
t
```

4. 属性监视器

属性监视器用来监控和响应属性值的变化。每次属性被设置值的时候，都会调用属性监视器，哪怕是新的值和原先的值相同。一个属性监视器由 `willSet` 和 `didSet` 组成，其定义形式如下：

```
var 属性名:数据类型=初始值{
    willSet(参数名){
        ...
    }
    didSet(参数名){
        ...
    }
}
```

其中，`willSet` 在设置新的值之前被调用，它会将新的属性值作为固定参数传入。`didSet` 在新的值被设置之后被调用，会将旧的属性值作为参数传入，可以为该参数命名或者使用默认参数名 `oldValue`。

【示例 4-7】 以下将使用属性监视器监视 `totalSteps` 属性值的变化。代码如下：

```
class StepCounter {
    var totalSteps: Int = 0 {
        //完整的属性监视器
        willSet(newTotalSteps) {
            println("新的值为 \(newTotalSteps)")
        }
        didSet(old) {
            if totalSteps > old {
                println("与原来相比增减了 \(totalSteps - old) 个值")
            }
        }
    }
}

let stepCounter = StepCounter()
stepCounter.totalSteps = 0
stepCounter.totalSteps = 5
stepCounter.totalSteps = 8
stepCounter.totalSteps = 16
```

此时运行程序，会看到如下的结果：

```
新的值为 0
新的值为 5
与原来相比增减了 5 个值
新的值为 8
与原来相比增减了 3 个值
新的值为 16
与原来相比增减了 8 个值
```


🔔注意：一个完整的属性监视器由 willSet 和 didSet 组成，但是 willSet 和 didSet 也可以单独使用。

4.1.4 方法

方法其实就是函数，用来实现某一特定功能的，只不过它被定义在了类中。在 Swift 中，方法分为实例方法和类型方法两种。以下就是对这两种方法的详细介绍。

1. 实例方法

实例方法是被类的实例化对象所调用的。实例方法和函数一样，分为不带参数和带参数两种。以下依次讲解这两种方法的使用。

(1) 不带参数的实例方法

不带参数的实例方法定义和函数的是一样的，其语法形式如下：

```
func 方法名() ->返回值类型{
    ...
}
```

调用形式如下：

```
对象名.方法名()
```

其中，对象名必须代表的是方法所属类的实例。

【示例 4-8】 以下遍历输出字符串"Hello"。代码如下：

```
class NewClass{
    var str="Hello"
    //实例方法
    func printHello(){
        for index in str {
            println(index)
        }
    }
}
var newvclass=NewClass()
newvclass.printHello() //调用
```

此时运行程序，会看到如下的结果：

```
H
e
l
l
o
```

(2) 具有参数的实例方法

具有参数的实例方法就是在方法名后面的括号中添加了参数列表。它的定义也和函数一样，定义形式如下：

```
func 方法名(参数 1:数据类型, 参数 2:数据类型, ...) ->返回值类型{
    ...
}
```

它的调用形式如下：

对象名.方法名(参数1,参数2,...)

【示例 4-9】 以下将输出指定范围的数字。代码如下：

```
class NewClass{
    //实例方法
    func rang(start:Int,end:Int){
        var i=start
        for(i;i<=end;++i){
            println(i)
        }
    }
}
var newclass=NewClass()
newclass.rang(0, end: 5) //调用
```

此时运行程序，会看到如下的结果：

```
0
1
2
3
4
5
```


2. 类型方法

类型方法是被类自身调用的方法。类型方法的定义形式如下：

```
class func 方法名(参数1:数据类型, 参数1:数据类型,...) {
    ...
}
```

类型方法调用的一般形式如下：

类名.方法名(参数1,参数2,...)

 **注意：** 这里的方法名为类型方法名。

类型方法和实例方法一样，可以分为无参数的类型方法和有参数的类型方法。不带参数列表的类型方法就是方法名后面的参数列表中没有参数。有参数的类型方法是具有参数列表方法。

【示例 4-10】 以下将在类中定义两个方法，其中一个方法实现字符串"Swift"的输出，另一个方法实现求和的功能。代码如下：

```
class NewClass{
    //无参数的类型方法
    class func printSwift(){
        println("Swift")
    }
    //具有参数的类型方法
    class func add(value1:Int,value2:Int)->Int{
        return value1+value2
    }
}
```



```

}
NewClass.printSwift()
println(NewClass.add(20, value2: 40))

```

此时运行程序，会看到如下的结果：

```

Swift
60

```

4.1.5 下标脚本

下标脚本是访问对象、集合或者序列的快速方式。开发者不需要调用实例特定的赋值和访问方法，就可以直接访问所需要的数值。下标脚本通过 subscript 关键字进行定义，其定义形式如下：

```

subscript(参数名称 1:数据类型, 参数名称 2:数据类型, ...) ->返回值的数据类型 {
    get {
        // 返回与参数类型匹配的类型的值
    }
    set(参数名称) {
        // 执行赋值操作
    }
}

```

在定义下标脚本时，需要注意以下两点。

- (1) set 参数名称必须和下标脚本定义的返回值类型相同，所以不为它指定数据类型。与计算属性相同，set 后面如果没有声明参数，那么就使用默认的 newValue。
- (2) 下标脚本可以和计算属性一样，可以设置为只读。只读的一般语法形式如下：

```

subscript(参数名称:数据类型) -> Int {
    get {
        //返回与参数匹配的 Int 类型的值
    }
}

```

可以简写为以下的形式：

```

subscript(参数名称:数据类型) -> Int {
    // 返回与参数匹配的 Int 类型的值
}

```

下标脚本的调用形式如下：

```
实例对象[参数 1, 参数 2, ...]
```

其中，[]和它里面的内容就代表了在类中定义的下标脚本。下标脚本可以分为具有一个传入参数的下标脚本和具有多个传入参数的下标脚本。

1. 具有一个传入参数的下标脚本

在数组中对某一个元素进行访问就使用了具有一个传入参数的下标脚本。此脚本最为常见。

【示例 4-11】 以下将实现对 3 门成绩求和的计算。代码如下：


```

class Score{
    var english:Int=100
    var chinese:Int=80
    var math:Int=70
    //定义下标脚本
    subscript(index:Int)->Int{
        get{
            switch index{
                case 0:
                    return english
                case 1:
                    return chinese
                case 2:
                    return math
                default:
                    return 0
            }
        }
    }
}
var myscore=Score()
var sum:Int=0
var i:Int=0
//遍历
for i=0;i<3;++i{
    sum+=myscore[i]
}
println(sum)

```

此时运行程序，会看到如下的结果：

250

2. 具有多个传入参数的下标脚本

具有多个传入参数的下标脚本一般会使用在多维数组中。

【示例 4-12】 以下就是使用具有两个参数的下标为二维数组赋值。代码如下：

```

class NewClass{
    let rows: Int = 0, columns: Int=0
    var grid: [Double]
    //初始化方法
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(count: rows * columns, repeatedValue: 0.0)
    }
    func isValidForRow(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    //下标脚本
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(isValidForRow(row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(isValidForRow(row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
        }
    }
}

```

```
    }  
}  
var matrix = NewClass(rows: 2, columns: 2)  
println("没有赋值前")  
println(matrix[0,0])  
println(matrix[0,1])  
println(matrix[1,0])  
println(matrix[1,1])  
println("赋值后")  
matrix[0,0]=1.0  
matrix[0,1]=5.6  
matrix[1,0]=2.4  
matrix[1,1]=3.2  
println(matrix[0,0])  
println(matrix[0,1])  
println(matrix[1,0])  
println(matrix[1,1])
```

此时运行程序，会看到如下的结果：

```
没有赋值前  
0.0  
0.0  
0.0  
0.0  
赋值后  
1.0  
5.6  
2.4  
3.2
```

4.1.6 类的嵌套

类的嵌套其实就是指在一个类中可以嵌套一个或者多个类。类的嵌套分为两种：直接嵌套和多层嵌套。下面依次讲解这两种方式。

1. 直接嵌套

直接嵌套就是将一个类或者多个类直接嵌套到另一个类中，形式如图 4.1 所示。

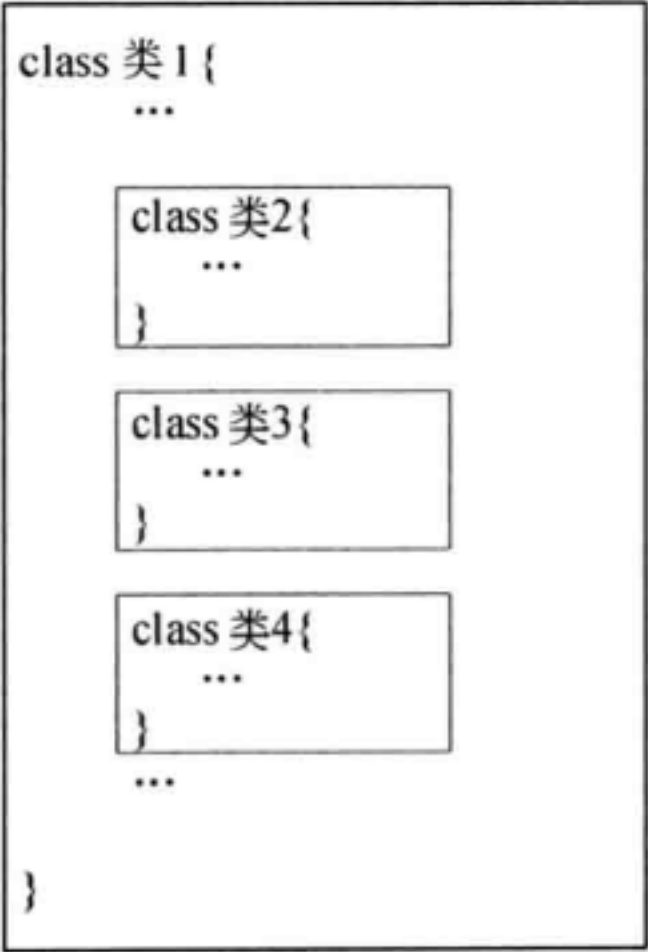


图 4.1 直接嵌套

在图 4.1 中, 类 2、类 3 和类 4 都是直接嵌套在类 1 中的。对于这种情况, 使用类 1 的实例属性和方法, 语法形式如下:

```
类 1().属性
类 1().方法
```

使用类 2 的实例属性和方法, 语法形式如下:

```
类 1.类 2().属性
类 1.类 2().方法
```

类 3 和类 4 的使用方法与之类似。

【示例 4-13】 以下将输出指定的字符串。代码如下:

```
class NewClass1 {
    class func printstr(str:String){
        println(str)
    }
    class NewClass2{
        class var str:String{
            return "Swift"
        }
    }
    class NewClass3{
        class var str:String{
            return "Hello"
        }
    }
    class NewClass4{
        class var str:String{
            return "World"
        }
    }
}
//调用
NewClass1.printstr(NewClass1.NewClass2.str)
NewClass1.printstr(NewClass1.NewClass3.str)
NewClass1.printstr(NewClass1.NewClass4.str)
```

此时运行程序, 会看到如下的结果:

```
Swift
Hello
World
```

在此代码中可以看到, NewClass1 中分别嵌套了 NewClass2、NewClass3 和 NewClass4 这 3 个类, 从而形成了直接嵌套。

2. 多层嵌套

多层嵌套的形式如图 4.2 所示。

在此图中可以看到类 3 和类 4 是直接嵌套在类 2 中的, 而类 2 又直接嵌套在类 1。这样形成了多层嵌套。这时, 如果访问类 1 的实例属性和方法, 其语法形式如下:

```
类 1().属性
类 1().方法
```

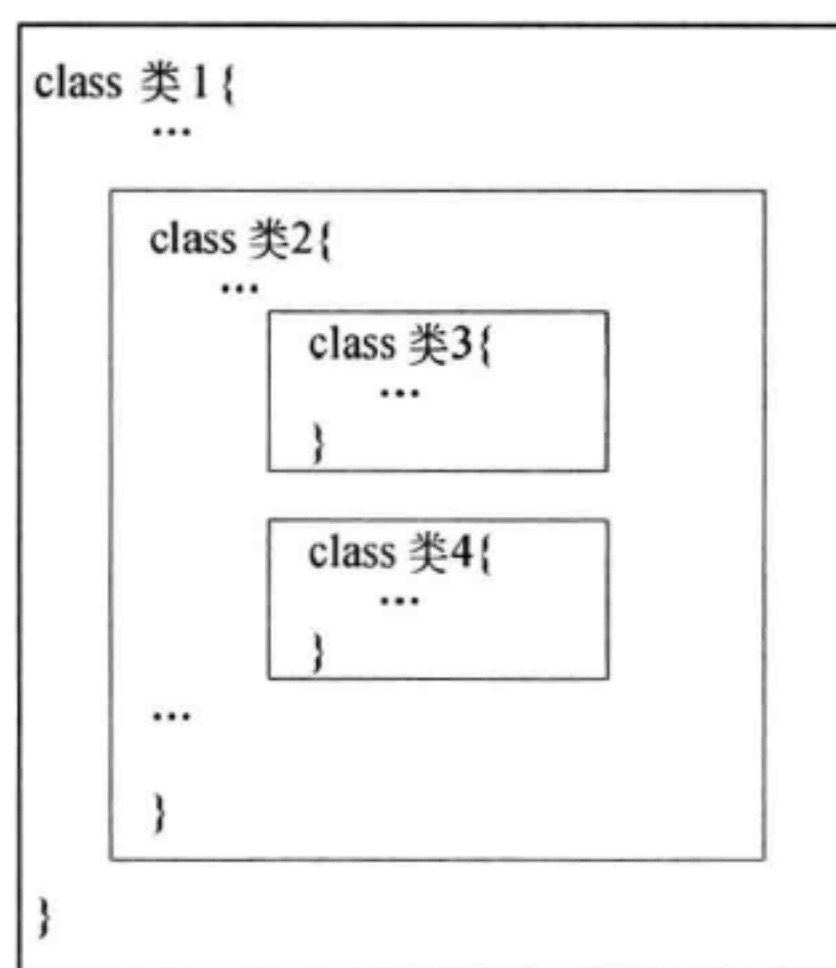



图 4.2 多次嵌套

如果要访问类 2 的实例属性和方法，对应的语法形式如下：

类 1.类 2().属性
类 1.类 2().方法

如果要访问类 3 的实例属性和方法，对应的语法形式如下：

类 1.类 2.类 3().属性
类 1.类 2.类 3().方法

如果要访问类 4 的实例属性和方法，对应的语法形式如下：

类 1.类 2.类 4.属性
类 1.类 2.类 4.方法

【示例 4-14】 以下是输出指定的字符串。代码如下：

```

class NewClass {
    class func printstr(str:String){
        println(str)
    }
    class StrClass{
        class var str:String{
            return "China"
        }
        class Str1Class{
            class var str:String{
                return "Hello"
            }
        }
        class Str2Class{
            class var str:String{
                return "Swift"
            }
        }
        class Str3Class{
            class var str:String{
                return "World"
            }
        }
    }
}

```

```

    }
}
//调用
NewClass.printstr(NewClass.StrClass.str)
NewClass.printstr(NewClass.StrClass.Str1Class.str)
NewClass.printstr(NewClass.StrClass.Str2Class.str)
NewClass.printstr(NewClass.StrClass.Str3Class.str)

```

此时运行程序，会看到以下的结果：

```

China
Hello
Swift
World

```

在此代码中可以看到，类 Str1Class、Str2Class、Str3Class 是直接嵌套在类 StrClass 中，而类 Str1Class 又直接嵌套在类 NewClass 中。这样就形成了多层嵌套。

4.1.7 可选链接

可选链接可以判断请求或调用的目标（属性、方法、下标脚本等）是否为空。如果目标有值，那么调用就会成功；相反，则返回空（nil）。对于多次请求或调用的可以被连接在一起形成一个链条。可选链接其实就是使用?运算符对可选类型实现的一种运算。开发者可以在想要调用的属性、下标脚本和方法的可选值后面添加一个?来进行可选链接的定义。以下就是对于这些可选链接的定义形式：

属性名?	//属性的可选链接
下标脚本?	//下标脚本的可选链接
方法名?	//方法的可选链接

对象可选链接的调用形式如下：

对象名.可选链接

【示例 4-15】 以下将在类中定义属性可选链接。代码如下：

```

class Residence {
    //定义一个可选类型的类型属性 numberOfRooms
    class var numberOfRooms:Int?{
        return 100
    }
    var number:Int?=nil
}
let newClass=Residence()
if let a=Residence.numberOfRooms? { //调用可选链接
    println("目标有值")
}else{
    println("目标为空")
}
if let a=newClass.number? { //调用可选链接
    println("目标有值")
}else{
    println("目标为空")
}

```

此时运行程序，会看到如下的结果：

目标有值
目标为空

开发者可以使用可选链接的可选值来调用属性、下标脚本和方法，并检查这些内容调用是否成功，它的调用形式如下：

可选链接.属性名	//调用属性
可选链接.方法	//调用方法
可选链接.[下标]	//调用下标脚本

【示例 4-16】 以下将通过自判断可选链接来调用属性值，并获取这个属性值。代码如下：

```
class Person {
    var residence: Residence?=nil
}
class Residence {
    var numberOfRooms = 10
}
let john1 = Person()
if let roomCount = john1.residence?.numberOfRooms { //通过可选链接调用属性
    println("John 在房子中有 \(roomCount) 个房间")
} else {
    println("无法检索房间数")
}
let john2 = Person()
let johnResidence = Residence()
john2.residence=johnResidence
if let roomCount = john2.residence?.numberOfRooms { //通过可选链接调用属性
    println("John 在房子中有 \(roomCount) 个房间")
} else {
    println("无法检索房间数")
}
```

此时运行程序，会看到如下的结果：

无法检索房间数
John 在房子中有 10 个房间

4.2 继 承

继承是使用已存在的类作为基础，建立新类的技术。新类的定义可以增加新的数据或新的功能。继承是面向对象的基本特性。通过继承的方式，可以实现类代码的高效重用。在大型项目中，使用继承可以节省大量的开发时间。本节将主要讲解类的实现、重写和类型检测等内容。

4.2.1 继承的实现

继承是使用:和一个类名来实现的，其语法形式如下：


```
class 子类名:父类名{
    ...
}
```

其中,冒号:是用来实现继承的。它表示子类继承了父类的特性。子类名要符合标识符的命名规则。父类必须在已经存在的类中。以下代码就是让 Fruits 继承 Food。

```
class Food{
    ...
}
class Fruits:Food{
    ...
}
```

其中, Food 就是父类; Fruits 就是子类, Fruits 继承了类 Food。一个子类一旦继承了父类,就说明在父类中声明的属性、方法和下标脚本等也被继承了。这时,在子类中可以使用父类的属性、方法,以及下标脚本。

【示例 4-17】 以下将定义两个类,分别为 NewClass1 和 NewClass2,让 NewClass2 继承 NewClass1,然后让 NewClass2 使用父类即 NewClass1 中的属性、下标脚本以及方法。代码如下:

```
class NewClass1{
    let value=100 //存储属性
    //计算属性
    var clastr:String{
        return "Calculate property"
    }
    //类型属性
    class var tystr:String{
        return "Type property"
    }
    var english:Int=100
    var chinese:Int=89
    var math:Int=98
    //定义下标脚本
    subscript(index:Int)->Int{
        switch index{
            case 0:
                return english
            case 1:
                return chinese
            case 2:
                return math
            default:
                return 0
        }
    }
}
//实例方法,功能是输出属性值 value
func printvalue(){
    println(value)
}
//类型方法,功能为输出字符串
class func printTypeFunc(){
    println("Type Func")
}
}
```

```
//定义 NewClass2, 作为 NewClass1 的子类
class NewClass2:NewClass1{
}
let newClass2=NewClass2() //实例化 NewClass2 类实例 newClass2
//输出 newClass2 中的属性内容
println(newClass2.value)
println(newClass2.clastr)
println(NewClass2.tystr)
//调用下标脚本, 并输出
println(newClass2[2])
println(newClass2[10])
//使用父类中的方法
newClass2.printvalue()
NewClass2.printTypeFunc()
```

此时运行程序, 会看到如下的结果:

```
100
Calculate property
Type property
98
0
100
Type Func
```

4.2.2 重写

重写就是子类对继承自父类的属性、下标脚本, 以及方法进行一些修改, 从而满足特定需求。以下是对属性、下标脚本, 以及方法的重写。

1. 重写属性

在类中的属性分为存储属性、计算属性, 以及类型属性。开发者可以对计算属性, 以及类属性进行重写。

(1) 计算属性

重写计算属性的语法形式如下:

```
override var 属性名:数据类型{
    get{
        ...
        return 某一属性值
    }
    set{
        ...
        属性值=某一个值
        ...
    }
}
```

其中, get 和 set 的部分是和计算属性中 get 和 set 的部分是一样的。

(2) 类型属性

重写类型属性的语法形式如下:


```
override class var 属性名:数据类型{
    ...
    返回一个值
}
```

2. 重写下标脚本

重写下标脚本的语法形式如下：

```
override subscript(入参参数列表) ->返回值的数据类型 {
    get {
        // 返回与入参匹配的类型值
    }
    set(newValue) {
        // 执行赋值操作
    }
}
```

3. 重写方法

在类中方法分为了两种：一种是实例方法；另一种是类型方法。以下就是对这两种方法进行重写的介绍。

(1) 实例方法

重写实例方法的语法形式如下：

```
override func 方法名(参数列表) ->返回值类型{
    ...
}
```

(2) 类型方法

重写类型方法的语法形式如下：

```
override class func 方法名(参数列表) ->返回值类型{
    ...
}
```

【示例 4-18】 以下将实现属性、下标脚本，以及方法的重写。代码如下：

```
class NewClass1{
    //计算属性
    var speed:Double{
        return 10
    }
    //类型属性
    class var str:String{
        return "Hello"
    }
    //下标脚本
    subscript(index:Int)->Int{
        var count=index-5
        return count
    }
    //实例方法，实现字符串"World"的输出
    func printstr(){
        println("World")
    }
}
```



```

//类型方法，实现两个数的减法运算
class func rang(start:Int,end:Int){
    println("两数相差\(end-start)")
}
}
class NewClass2:NewClass1{
    //重写计算属性
    override var speed:Double{
        return 10
    }
    //重写类型属性
    override class var str:String{
        return "Swift"
    }
    //重写下标脚本
    override subscript(index:Int)->Int{
        var count=index+100
        return count
    }
    //重写实例方法，实现输出字符串("I Love Swify")
    override func printstr() {
        println("I Love Swify")
    }
    //重写类型方法，实现范围中数字的输出
    override class func rang(start:Int,end:Int){
        var i=start
        for(i;i<=end;++i){
            println(i)
        }
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var newclass2=NewClass2()
        println(newclass2.speed) //输出重写的计算属性
        println(NewClass2.str) //输出重写的类型属性
        println(newclass2[8]) //调用重写的下标脚本
        newclass2.printstr() //调用重写的实例方法
        NewClass2.rang(0, end: 5) //调用重写的类型方法
    }
}

```

此时运行程序，会看到以下的结果：

```

10.0
Swift
108
I Love Swify
0
1
2
3
4
5

```

4. 添加属性监视器

开发者可以为继承来的属性添加属性监视器，这样一来，当继承来得属性值发生改变

时, 开发者就会被通知到, 其语法形式如下:

```
override var 属性名:数据类型=初始值{
    willSet(参数名){
        ...
    }
    didSet{
        ...
    }
}
```

【示例 4-19】 以下为继续来的属性 speed 添加属性监视器。代码如下:

```
class Car{
    var speed: Double = 0.0
}
//定义类 AutomaticCar, 作为 Car 的子类
class AutomaticCar: Car {
    var gear = 1
    override var speed: Double {
        //添加属性监视器
        didSet {
            gear = Int(speed)
            println(gear)
        }
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let automatic = AutomaticCar()
        automatic.speed = 35.0
        automatic.speed = 100.0
        automatic.speed = 300.0
    }
}
```

此时运行程序, 会看到如下的结果:

```
35
100
300
```

📌注意: 在子类中重写了父类中的属性、下标脚本和方法后, 有时为了满足某种需求, 需要访问父类中相对应的部分。此时就需要使用 `super` 来实现。以下就是对父类中的属性、下标脚本和方法进行访问的语法形式。

<code>super.属性名</code>	//访问父类的属性
<code>super[下标]</code>	//访问父类的下标脚本
<code>super.方法名(参数值)</code>	//访问父类的方法

其中, `super` 表示父类。

【示例 4-20】 以下将以【示例 4-18】为基础, 实现对父类的访问。代码如下:

```
class NewClass2: NewClass1 {
    .....
    func superclass1(){
        println(super.speed) //访问父类的计算属性
    }
}
```



```

        println(super[5])           //访问父类的下标脚本
        super.printstr()           //访问父类的实例方法
    }
    class func superclass2(){
        super.str                   //访问父类的类型属性
        super.rang(5, end: 20)      //访问父类的类型方法
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var newclass2=NewClass2()
        newclass2.superclass1()
        NewClass2.superclass2()
    }
}

```

此时运行程序，会看到如下的结果：

```

10.0
0
World
两数相差 15

```

4.2.3 禁止重写

虽然在父类中的属性、下标脚本，以及方法可以在子类中重写，但是有的时候需要禁止对某一个属性、下标脚本或者方法进行重写。此时，就需要使用 **final** 关键字来禁止这些内容的重写。当这些内容被禁止重写后，是不允许再在子类中进行重写的。否则就会出现错误。

【示例 4-21】 以下将使用 **final** 关键字让计算属性 **value** 禁止重写，但是可以在 **NewClass1** 的子类中重写的此属性。代码如下：

```

class NewClass1{
    final var value:Int{
        return 100
    }
}
class NewClass2:NewClass1{
    //重写类型方法
    override var value:Int{
        return 50
    }
}

```

由于在父类中使用了 **final** 阻止了计算属性 **value** 的重写，但在此代码中又在子类中重写了计算属性导致程序出现了以下的错误：

```
Var overrides a 'final' var
```

4.2.4 类型检测

类型检查是一种检查类实例的方式。在 **Swift** 中提供了类型检查操作符：一个是 **is** 操

作符，另一个是 as 操作符。本节将对这两个操作符进行详细介绍。

1. is检测符

is 检测符可以用来检查一个实例（实例对象）是否属于特定的子类型。其语法形式如下：

实例 is 子类型

其中，它的返回值类型为布尔类型。当返回值为 true 时，表示实例属于子类型；当为 false 时，表示不属于子类型。

【示例 4-22】 以下将使用 is 检测符进行判断指定的实例是否属于某一子类。代码如下：

```
class NewClass1{
}
//定义类 NewClass2, 作为 NewClass1 的子类
class NewClass2:NewClass1{
}
//定义类 NewClass3, 作为 NewClass1 的子类
class NewClass3:NewClass1{
}
let library=[NewClass2(),NewClass2(),NewClass2(),NewClass3(),NewClass2(),
NewClass2()]
var newclass1Count=0
var newclass2Count=0
//遍历数组
for item in library{
    //判断当前实例是否属于 NewClass2
    if item is NewClass2{
        ++newclass1Count
    }
    //判断当前实例是否属于 NewClass3
    else if item is NewClass3{
        ++newclass2Count
    }
}
println("NewClass2 的实例有 \(newclass1Count) 个")
println("NewClass3 的实例有 \(newclass2Count) 个")
```

此时运行程序，会看到如下的结果：

```
NewClass2 的实例有 5 个
NewClass3 的实例有 1 个
```

2. as检测符

as 检测符一般用于在程序运行期间进行类型转换。它分为两种类型：一种是强制转换形式的检查符 as，另一种是可选转换形式的类型检查符 as?。以下就是对这两种类型的介绍。

(1) 强制转换形式的检查

as 检查符是当转换一定成功时，才可以使用的，其语法形式如下：

实例 as 类型

【示例 4-23】 以下将使用强制转换形式的 `as` 检测符实现类型的转换。代码如下：

```
class NewClass1{
}
//定义 NewClass2 类，作为 NewClass1 的子类
class NewClass2:NewClass1{
    var str: String="Swift"
}
let library = NewClass2()
let class2=library as NewClass2                                //转换
println("NewClass2: \(class2.str)")
```

此时运行程序，会看到如下的结果：

```
NewClass2: Swift
```

(2) 可选转换形式的类型检查

当开发者不确定向下转换是否可以成功时，可以使用可选转换形式的类型检查——`as?`。其语法形式如下：


实例 `as?` 类型

【示例 4-24】 以下将使用可选转换形式的类型检查 `as?`检测符实现类型的转换。代码如下：

```
class NewClass1{
}
//定义类 NewClass2，作为 NewClass1 的子类
class NewClass2:NewClass1{
    var str: String="NewClass2"
}
//定义类 NewClass3，作为 NewClass1 的子类
class NewClass3:NewClass1{
    var str: String="NewClass3"
}
let library = [NewClass2(),NewClass3(),NewClass3()]
//遍历
for item in library{
    //判断是否转为 NewClass2
    if let class2=item as?NewClass2{
        println(class2.str)
    }
    //判断是否转为 NewClass3
    }else if let class3=item as? NewClass3 {
        println(class3.str)
    }
}
```

此时运行程序，会看到如下的结果：

```
NewClass2
NewClass3
NewClass3
```

 **注意：**对于一些类型不明确的类型，Swift 提供了两种类型别名来检查，分别为 `AnyObject` 和 `Any`。其中，`AnyObject` 可以代表任何类型的实例，而 `Any` 比 `AnyObject` 使用的范围更广，它可以表示任何类型，除了方法类型。

【示例 4-25】 以下将使用 Any 类型别名定义一个数组，然后使用 as 进行类型检查。代码如下：

```
class NewClass1{
    var director:String{
        return "AAA"
    }
}
//定义类 NewClass2, 作为 NewClass1 的子类
class NewClass2:NewClass1{
    override var director:String{
        return "Orson Welles"
    }
}
//向数组 things 中添加元素
var things = [Any]()
things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append(NewClass1())
things.append(NewClass2())
//遍历
for thing in things {
    switch thing {
    case 0 as Int:
        println("0 是整型")
    case 0.0 as Double:
        println("0.0 是浮点型")
    case let someInt as Int:
        println("\(someInt) 是一个整型值")
    case let someDouble as Double where someDouble > 0:
        println("\(someDouble) 可能是一个浮点型值")
    case let someString as String:
        println("\(someString) 是一个字符串")
    default:
        println("其它")
    }
}
```

此时运行程序，会看到如下的结果：

```
0 是整型
0.0 是浮点型
42 是一个整型值
3.14159 可能是一个浮点型值
hello 是一个字符串
其它
其它
```

4.3 枚 举

枚举定义了一组具有相关性的数据，如常量和字符串等，使开发者可以在代码中以一

个安全的方式来使用这些值，它们可以有助于提供代码的可读性。本节将主要讲解枚举的定义、成员的定义、实例化对象和访问原始值等内容。


4.3.1 定义枚举

在 Swift 中，枚举可以分为两种：一种是任意类型的枚举（没有指定数据类型）；另一种是指定数据类型的枚举类型。以下就是这两种枚举类型的定义。

1. 任意类型的枚举类型

任意类型的枚举就是没有指定数据类型的枚举，它的定义需要使用到关键字 `enum`，其语法形式如下：

```
enum 枚举名称 {
    ...
}
```

 **注意：**枚举名称要符合标识符的命名规则。此时的枚举类型是一个具有任意类型的类型。

2. 指定数据类型的枚举类型

指定数据类型的枚举类型也是使用 `enum` 关键字进行定义的，但是要在枚举名称后面添加冒号以及数据类型。其定义形式如下：

```
enum 枚举名称:数据类型 {
    ...
}
```

【示例 4-26】 以下将分别定义任意类型的枚举和指定数据类型的枚举，代码如下：

```
enum NewEnum1 {
}
enum NewEnum2:String {
}
```

在此代码中，`NewEnum1` 枚举是没有指定数据类型的枚举，它是任意类型的枚举；`NewEnum2` 枚举是一个字符串类型的枚举。

4.3.2 定义枚举成员

根据枚举的类型的不同，枚举成员的定义也分为两种：一种是定义任意类型的枚举成员；另一种是定义指定类型的枚举成员。以下是对这两种定义枚举成员的介绍。

1. 定义任意类型的枚举成员

定义任意类型的枚举成员需要使用关键字 `case`，其语法形式如下：

```
enum 枚举名称 {
    case 成员名称 1
    case 成员名称 2
}
```

```
...
}
```

其中，关键字 `case` 表示成员中一条新的分支被定义。在定义任意类型的枚举成员时，需要注意以下两点。

(1) 在具有任意类型的枚举类型中，可以将它里面的成员定义为不同类型，其语法形式如下：

```
enum 枚举名称{
    case 成员名 1(数据类型)
    case 成员名 2(数据类型)
    ...
}
```

(2) 在任意类型中，可以不定义成员，从而形成一个空的枚举类型。

2. 定义指定类型的枚举成员

定义指定数据类型的枚举类型成员也同样需要使用 `case`，其语法形式如下：

```
enum 枚举名称:数据类型 {
    case 成员名称 1=初始值
    case 成员名称 2=初始值
    ...
}
```

在定义指定类型的枚举成员时，需要注意以下 4 点。

(1) 在指定数据类型的枚举类型中，如果原始类型是整型，可以不指定初始值。如果指定的类型并非整型，需要指定初始值。

(2) 在为枚举类型指定数据类型后，它里面的成员也都为此数据类型，不可以定义其他类型。

(3) 在为成员指定初始值时，初始值必须是唯一标识的。

(4) 在指定数据类型的枚举类型中，必须使用 `case` 关键字定义成员。


4.3.3 实例化枚举的对象

在使用枚举之前，首先要对枚举进行实例化，这一点和类是一样的。枚举的实例化的语法形式如下：

```
let/var 枚举对象名=枚举类型.成员名
```

或者：

```
枚举类型.成员名
```

 **注意：**枚举类型的实例化必须要使用成员，而不是其他。

【示例 4-27】 以下将定义一个枚举类型，然后再进行实例化。代码如下：

```
enum Parterre:String{
    case nickname1="Rose"
```



```

    case nickname2="Carnation"
    case nickname3="tulip"
    case nickname4="orchid"
    case nickname5="peony"
    case nickname6="cactus"
}
let flower=Parterre.nickname1 //实例化
println(flower)

```

此时运行程序，会看到如下的结果：

```
(Enum Value)
```

⚠注意：(Enum Value)表示的是 nickname1 这个成员，而不是原始值。

4.3.4 枚举成员与 switch 语句的匹配

由于枚举具有多个成员，相当于 switch 中的分支结构。所以枚举经常匹配 switch 语句进行使用，从而可以将成员的原始值进行获取。

【示例 4-28】 以下将使用 Parterre 枚举与 switch 语句进行匹配。代码如下：

```

enum Parterre:String{
    case nickname1="Rose"
    case nickname2="Carnation"
    case nickname3="Tulip"
    case nickname4="Orchid"
    case nickname5="Peony"
    case nickname6="Cactus"
}
let flower=Parterre.nickname5
switch flower{
    case .nickname1:
        println("Rose")
    case .nickname2:
        println("Carnation")
    case .nickname3:
        println("Tulip")
    case .nickname4:
        println("Orchid")
    case .nickname5:
        println("Peony")
    case .nickname6:
        println("Cactus")
}

```

此时运行程序，会看到如下的结果：

```
Peony
```

⚠注意：将枚举成员和 switch 语句进行匹配时，switch 语句必须要和每一个枚举成员进行匹配，不可以有遗漏。如果开发者不需要匹配每一个枚举成员时，可以提供一個默认的 default 分支来涵盖所有没有明确被匹配的成员。

4.3.5 访问枚举中成员的原始值

使用枚举匹配 `switch` 语句可以用来获取或者访问成员值，除此之外，Swift 还提供了两个访问原始值的方法：一个是 `toRaw()`；另一个是 `fromRaw()`。

1. `toRaw()`

`toRaw()`方法可以通过枚举中的成员对相应的原始值进行访问。其语法形式如下：

```
let/var 常量名/变量名=枚举名称.成员名称.toRaw()
```

【示例 4-29】 以下将使用 `toRaw()`方法实现原始值的访问。代码如下：

```
enum Digital:Int{
    case one=1
    case two=2
    case three=3
    case four=4
    case five=5
}
let onevalue=Digital.one.toRaw()           //访问成员 one 的原始值
println(onevalue)
let fivevalue=Digital.five.toRaw()         //访问成员 two 的原始值
println(fivevalue)
```

此时运行程序，会看到如下的结果：

```
1
5
```

在访问指定类型为整型的枚举的成员时，需要注意以下两点。

(1) 如果有其他的成员没有指定原始值，那么它们会自动递增。即如果第二个成员的原始值为 5，那么第 3 个成员的原始值为 6，第 4 个成员的原始值为 7，依此类推。

(2) 如果所有的成员都没有指定原始值的话，那么在枚举中第一个成员的原始值就为 0，然后再自动递增，即第二个成员的原始值为 1，第 3 个成员的原始值为 2，依此类推。

2. `fromRaw()`

`fromRaw()`方法正好与 `toRaw()`方法的功能相反。它的功能是通过原始值来获取成员。其语法形式如下：

```
let/var 常量名/变量名 =枚举类型名.fromRaw(原始值)
```

其中，此方法的类型是一个可选类型。

【示例 4-30】 以下将使用 `fromRaw()`获取枚举中的成员。代码如下：

```
enum Digital:Int{
    case one=1
    case two=2
    case three=3
}
//通过原始值获取成员
if let onemember=Digital.fromRaw(2){
```

```

let a=onemember
switch a{
    case .one:
        println("one")
    case .two:
        println("two")
    case .three:
        println("three")
}
}

```

此时运行程序，会看到如下的结果：

```
two
```

4.3.6 相关值

在任意类型的枚举类型中，可以定义任意的成员，这些成员是没有值的。在实例化时，可以定义此成员的相关信息，这些信息被称为相关值。相关值的定义形式：

```
let/var 枚举类型对象=枚举类型名.成员名(相关值)
```

【示例 4-31】 以下将为定义任意类型的枚举类型 Barcode 的成员添加相关值。代码如下：


```

enum Barcode {
    //定义了任意类型的成员
    case UPCA(Int, Int, Int)
    case QRCode(String)
}
var productBarcode = Barcode.UPCA(8, 85909_51226, 3) //相关值
switch productBarcode {
    case .UPCA(let numberSystem, let identifier, let check):
        println("UPC-A 的值有: \(numberSystem), \(identifier), \(check).")
    case .QRCode(let productCode):
        println("QR code 的值有: \(productCode).")
}

```

此时运行程序，会看到如下的结果：

```
UPC-A 的值有: 8, 8590951226, 3.
```

 **注意：**在以上的 switch 代码中我们使用了数值绑定，所谓使用绑定就是在 case 匹配的同时，可以将 switch 语句中的值绑定给一个指定的变量或者常量，以便在 case 语句中使用。

4.3.7 定义枚举类型的其他

在枚举类型中除了可以定义成员外，还可以和类一样定义属性、下标脚本和方法等。以下是对这些内容的讲解。

1. 属性

在枚举类型中只可以定义两种属性，即计算属性和类型属性。以下是这两种属性定义

的详细介绍。

(1) 计算属性

计算属性可以在枚举类型中定义。它的定义形式和在类中进行定义是一样的，调用形式如下：

```
枚举对象名.计算属性名
```

(2) 类型属性

在枚举中类型属性的定义方法和类中类型属性的定义方法是有区别的，其定义形式如下：

```
static var 类型属性名:数据类型{
    ...
}
```

它的调用形式和类调用类型属性是一样的，其语法形式如下：

```
枚举类型名.类型属性
```

2. 下标脚本

下标脚本在枚举中的定义和在类中定义是一样的。它的调用形式如下：

```
枚举对象名[下标值]
```

3. 方法

在枚举中定义的方法也分为两种：一种是实例方法；另一种是类型方法。以下是对这两种方法的介绍。

(1) 实例方法

实例方法在枚举中定义的语法形式如下：

```
func 方法名(参数名1:数据类型, ...) ->返回值类型{
    ...
}
```

调用的形式如下：

```
枚举类型名.成员名.方法名(参数1, ...)
```

(2) 类型方法

类型方法在枚举中定义的语法形式如下：

```
static func 方法名(参数名1:数据类型, ...) ->返回值类型{
    ...
}
```

类型方法的调用形式如下：

```
枚举类型名.方法名(参数1, ...)
```

【示例 4-32】 以下将在枚举中实现属性、下标脚本、方法的定义，以及调用。代码如下：


```

enum NewEnum: Int {
    case one = 1
    case two = 2
    // 计算属性
    var value1: Int {
        return 100
    }
    // 类型属性
    static var value2: Int {
        return 10000
    }
    // 下标脚本
    subscript(index: Int) -> String {
        switch index {
            case 1:
                return "one"
            case 2:
                return "two"
            default:
                return "没有对应的成员"
        }
    }
    // 实例方法
    func addValue(i: NewEnum, j: NewEnum) -> Int {
        var sum = i.rawValue + j.rawValue
        return sum
    }
    // 类型方法
    static func printstr() {
        println("NewEnum")
    }
}

let a = NewEnum.one
println(a.value1) // 调用计算属性
println(NewEnum.value2) // 调用类型属性
println(a[1]) // 下标脚本
println(a[5]) // 下标脚本
a.addValue(a, j: NewEnum.two) // 调用实例方法
NewEnum.printstr() // 调用类型方法


```

此时运行程序，会看到如下的结果：

```

100
10000
one
没有对应的成员
NewEnum

```

 **注意：**除了可以在枚举中定义属性、下标脚本以及方法外，还可以在其中添加属性监视器。

4.3.8 枚举的嵌套

枚举也可以嵌套使用。和类一样，枚举的嵌套也分为直接嵌套和多层嵌套。以下就是对这两种嵌套的介绍。

1. 直接嵌套

在一个枚举类型中可以嵌套一个或者多个枚举类型，其形式如下：

```
enum 枚举名称 1 {
    enum 枚举名称 2 {
        ...
    }
    enum 枚举名称 3 {
        ...
    }
}
```

在此形式中，枚举类型 2 和枚举类型 3 都是直接嵌套在枚举类型 1 中。对于这种情况，使用枚举类型 1 中的成员、实例属性和实例方法，其语法形式如下：

```
枚举类型 1.成员
枚举类型 1.成员.属性
枚举类型 1.成员.方法
```

使用枚举类型 1 中的类型属性和类型方法的语法形式如下：

```
枚举类型 1.属性
枚举类型 1.方法
```

使用枚举类型 2 中的成员、实例属性和实例方法，语法形式如下：

```
枚举类型 1.枚举类型 2.成员
枚举类型 1.枚举类型 2.成员.属性
枚举类型 1.枚举类型 2.成员.方法
```

使用枚举类型 2 中的类型属性和类型方法的语法形式如下：

```
枚举类型 1.枚举类型 2.属性
枚举类型 1.枚举类型 2.方法
```

2. 多层嵌套

多层嵌套就是在一个嵌套枚举类型中又嵌套了枚举类型。其形式如下：

```
enum 枚举名称 1 {
    enum 枚举名称 2 {
        enum 枚举名称 3 {
            ...
        }
        enum 枚举名称 4 {
            ...
        }
    }
}
```

枚举类型 3 和枚举类型 4 是直接嵌套在枚举类型 2 中，而枚举类型 2 又直接嵌套在枚举类型 1 中。这样形成了多层嵌套。这时，如果访问枚举类型 1 的成员、实例属性和实例方法，其语法形式如下：


```
枚举类型 1.成员  
枚举类型 1.成员.属性  
枚举类型 1.成员.方法
```

访问枚举类型 1 的类型属性和类型方法，对应的语法形式如下：

```
枚举类型 1.属性  
枚举类型 1.方法
```

如果要访问枚举类型 2 的成员、实例属性和实例方法，对应的语法形式如下：

```
枚举类型 1.枚举类型 2.成员  
枚举类型 1.枚举类型 2.成员.属性  
枚举类型 1.枚举类型 2.成员.方法
```

访问枚举类型 2 的类型属性和类型方法，对应的语法形式如下：

```
枚举类型 1.枚举类型 2.属性  
枚举类型 1.枚举类型 2.方法
```

如果要访问枚举类型 3 的成员、实例属性和实例方法，对应的语法形式如下：

```
枚举类型 1.枚举类型 2.枚举类型 3.成员  
枚举类型 1.枚举类型 2.枚举类型 3.成员.属性  
枚举类型 1.枚举类型 2.枚举类型 3.成员.方法
```

访问枚举类型 3 的类型属性和类型方法，对应的语法形式如下：

```
枚举类型 1.枚举类型 2.枚举类型 3.属性  
枚举类型 1.枚举类型 2.枚举类型 3.方法
```


4.4 结 构

在现实生活中，一个人需要有姓名、工作单位、E-mail 地址、联系电话等信息。人们通过采用名片的形式，将这些信息印在一张纸上，这样便于管理个人信息。而在编程中，人们使用结构来代替名片，同样可以达到一样的效果。结构（struct）是由一系列具有相同类型或不同类型的数据构成的数据集合，也叫结构体。结构体和类有很多相同的地方。但结构比类使用更简单，运行效率也更高。本节将会讲解结构的定义、实例化，以及定义属性、下标脚本方法等内容。

4.4.1 定义结构

结构是通过关键字 struct 进行定义的，其语法形式如下：

```
struct 结构名称{  
    ...  
}
```

注意：结构名称要符合标识符的命名规则。

4.4.2 实例化结构对象

结构的实例化对象和类的实例化对象相似，其语法形式如下：

```
let/var 变量名=结构体名称()
```

【示例 4-33】 以下将定义一个 `SomeStructure` 结构，然后再对此结构进行实例化。代码如下：

```
struct SomeStructure {  
    ...  
}  
let newstruct= SomeStructure ()
```

4.4.3 在结构中定义内容

和类一样，在结构中也是可以定义属性、下标脚本以及方法的。以下是对这些内容的介绍。

1. 属性

在类中我们知道属性包括 3 种：存储属性、计算属性以及类型属性。在结构中可以对这 3 种属性进行定义。以下是这 3 种属性定义和调用的形式。

(1) 存储属性

存储属性的定义和类是一样的，分为两种，一种是定义常量存储属性，另一种是定义变量存储属性，其中定义常量存储属性的语法形式如下：

```
let 常量存储属性:数据类型=初始值
```

定义变量存储属性的语法形式如下：

```
var 变量存储属性:数据类型=初始值
```

其中，数据类型是可以省略不写的。而对于结构对象的任何属性的访问和修改都是使用 `.来` 进行的。

```
结构对象.属性
```

(2) 计算属性

计算属性的定义形式和在类中或者枚举类型中的定义形式是一样的。其调用形式如下：

```
结构对象名.计算属性名
```

(3) 类型属性

类型属性在结构体中的定义形式如下：

```
static var 类型属性名:数据类型{  
    ...  
}
```

调用形式如下：


结构名称.类型属性

【示例 4-34】 以下将在结构中实现存储属性、计算属性，以及类型属性的定义和调用。代码如下：

```
struct SomeStructure{
    //定义存储属性
    let value1:Int=20
    //定义计算属性
    var value2:Int{
        return 500
    }
    //定义类型属性
    static var str: String {
        return "Struct"
    }
}
let newstruct=SomeStructure()
println(newstruct.value1)           //访问存储属性
println(newstruct.value2)           //访问计算属性
println(SomeStructure.str)           //访问类型属性
```

此时运行程序，会看到如下的结果：

```
20
500
Struct
```

 **注意：** 开发者除了可以在结构中定义属性外，还可以添加属性监视器。对于属性监视器在结构中的定义形式和在类中定义形式是一样的。

2. 下标脚本

在结构中定义下标脚本，它的定义和在类或者在枚举中的定义是一样的，调用形式如下：

结构对象名[下标值]

【示例 4-35】 以下将实现下标脚本计算 3 门成绩的平均值。代码如下：

```
struct Score {
    let Math=98
    let English=100
    let Chinese=60
    //定义下标脚本
    subscript(index: Int) -> Int {
        switch index{
            case 0:
                return Math
            case 1:
                return English
            case 2:
                return Chinese
            default:
                return 0
        }
    }
}
```



```

    }
}
let value=Score()
var i=0
var results=0
//遍历访问
for i;i<3;++i{
    results+=value[i]
}
println("三门课的成绩的平均值=\(results/3)")

```

此时运行程序，会看到如下的结果：

```
三门课的成绩的平均值=86
```

3. 方法

在结构中可以定义实例方法和类型方法。以下将详细介绍这两个方法。

(1) 实例方法

实例方法的定义和在类、枚举类型中的定义是一样的，其定义形式如下：

```

func 方法名(参数名1:数据类型,参数名2:数据类型,...){
    ...
}

```

实例方法的调用形式如下：

```
结构对象名.方法名(参数1,...)
```

(2) 类型方法

类型方法也可以定义在结构中，其语法形式如下：

```

static func 方法名(参数名1:数据类型,参数名2:数据类型,...){
    ...
}

```

它的调用形式如下：

```
结构名称.方法名(参数1,参数2,...)
```

【示例 4-36】 以下将在结构中定义两个方法，分别为实例方法和类型方法。其中，实例方法可以实现求最大值的功能，类型方法可以实现让数组中的元素进行倒序输出。代码如下：

```

struct SomeStructure{
    //定义实例方法，判断最大值
    func maxvalue(value1:Int,value2:Int)->Int{
        return max(value1, value2)
    }
    //定义类型方法，让数组中的元素倒序输出
    static func desc(value:[Int]){
        for i in reverse(value){
            println(i)
        }
    }
}

```



```
let newstruct=SomeStructure()
println(newstruct.maxvalue(100,value2:5))           //调用
let array=[1,0,0,8,6]
SomeStructure.desc(array)                          //调用
```

此时运行程序，会看到如下的结果：

```
100
6
8
0
0
1
```

4.5 构造方法和析构方法

在 Swift 中存在两种特殊的方法，即构造方法和析构方法。构造方法往往使用在对类型进行初始化时，一般被称为构造器。而析构方法使用在类中，在类实例被释放时调用，用来执行特定的清除工作。本节将主要讲解这两种特殊的方法。

4.5.1 值类型的构造器

值类型的构造器主要是在数值类型（数值类型包括结构和枚举）的实例进行初始化时所使用的。本小节将主要讲解值类型构造器的默认构造器、结构的逐一成员构造器、自定义构造器，以及构造器代理等内容。

1. 默认构造器

值类型的默认构造器将结构实例的属性都设为默认值。默认值都是在结构属性声明中指定的，其语法形式如下：

```
let/var 结构对象名=结构名称()
```

2. 结构的逐一成员构造器

除上面提到的默认构造器，如果结构对所有存储型属性提供了默认值且自身没有提供定制的构造器时，它们能自动获得一个逐一成员构造器。逐一成员构造器是用来初始化结构新实例里成员属性的快捷方法。我们在调用逐一成员构造器时，通过与成员属性名相同的参数名进行传值来完成对成员属性的初始赋值。其语法形式如下：

```
let/var 结构对象名=结构名称(属性名1:内容,属性名2:内容)
```

【示例 4-37】 以下将使用逐一成员构造器实例化结构。代码如下：

```
struct NewStruct{
    var width=20
    var height=66
}
let newstruct1=NewStruct()           //默认构造器
println("\(newstruct1.width), \(newstruct1.height)")
```

```
let newstruct2=NewStruct(width:500,height:100)           //逐一成员构造器
println("\(newstruct2.width), \(newstruct2.height)")
```

此时运行程序，会看到如下的结果：

```
20,66
500,100
```


3. 自定义构造器

在值类型中，除了可以有默认的构造器、逐一成员构造器外，还可以有自定义的构造器。自定义的构造器需要使用关键字 `init` 进行定义，其语法形式如下：

```
init(参数名1:数据类型,参数名2:数据类型,...){
    ...
}
```

自定义构造器的调用形式如下：

```
let/var 对象名=值类型的类型名(参数名1:内容,参数名2:内容,...)
```


 注意：在构造器中，参数名默认都是外部参数名。

【示例 4-38】 以下将在结构 `Color` 中定义具有 3 个参数的自定义构造器。代码如下：

```
struct Color {
    var red:Double
    var green:Double
    var blue:Double
    //带有 3 个参数的构造器
    init(Red: Double, Green: Double, Blue: Double) {
        red = Red
        green = Green
        blue = Blue
    }
}
let magenta = Color(Red: 1.0, Green: 0.0, Blue: 1.0)
println("red=\(magenta.red)")
println("green=\(magenta.green)")
println("blue=\(magenta.blue)")
```

此时运行程序，会看到如下的结果：

```
red=1.0
green=0.0
blue=1.0
```

 注意：在自定义构造器中，最简单的就是没有参数的自定义构造器，它的定义语法形式如下：

```
init(){
    ...
}
```


没有参数的自定义构造器的调用形式如下：

```
let/var 对象名=值类型的类型名()
```


4. 构造器代理

构造器代理其实就是通过调用其他构造器来完成实例的部分构造过程。它能减少多个构造器间的代码重复、减少编程时间、提高代码的可读性等。在值类型和类中都有构造器代理，但是由于值类型不支持继承，所以构造器代理的过程相对简单。构造器代理的语法形式如下：

```
init(参数名 1:数据类型) {
    ...
}
init(参数名 2:数据类型) {
    ...
    self.init(参数名 1:参数名 2)
    ...
}
```

 **注意：**以上这种语言形式是最简单的语言形式。参数名 1 和参数名 2 的数据类型相同。

【示例 4-39】 以下将在一个构造器中调用另一个具有参数的构造器。代码如下：

```
struct Size {
    var width = 0.0
    var height = 0.0
}
//定义 Point 结构
struct Point {
    var x = 0.0
    var y = 0.0
}
//定义 Rect 结构
struct Rect {
    var origin = Point()
    var size = Size()
    //自定义构造器
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    //自定义构造器
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
        //构造器代理
    }
}
let centerRect = Rect(center: Point(x: 4.0, y: 4.0), size: Size(width: 3.0, height: 3.0))
println("矩形的原点为: ")
println(centerRect.origin.x)
println(centerRect.origin.y)
println("矩形的尺寸为: ")
println(centerRect.size.width)
println(centerRect.size.height)
```

此时运行程序，会看到如下的结果：

矩形的原点为:

2.5

2.5

矩形的尺寸为:

3.0

3.0

4.5.2 类的构造器

类和值类型一样也是有构造器的。本小节将讲解类构造器的相关内容,如默认构造器、自定义构造器,以及构造器代理等内容。

1. 默认构造器

其实,类的默认构造器我们在对类进行实例化时就已经接触到了,它的语法形式如下:

```
let/var 对象名=类名()
```

2. 自定义构造器

在类中自定义构造器有两种:一种为指定构造器;另一种为便利构造器。以下是对这两种构造器的详细介绍。

(1) 指定构造器

指定构造器是类中最主要的构造器。它会初始化类中提供的所有属性。指定构造器需要使用 `init` 关键字进行定义,其语法形式如下:

```
init(参数名1:数据类型,参数名2:数据类型,...) {
    ...
}
```

指定构造器的调用形式如下:

```
let/var 对象名=类名(参数名1:内容,参数名2:内容,...)
```

【示例 4-40】 以下将实现指定构造器的定义以及调用。代码如下:

```
class NewClass{
    var value1:Int
    var value2:Float
    var value3:String
    //定义带有 3 个参数的指定构造器
    init(i:Int,j:Float,k:String){
        value1=i
        value2=j
        value3=k
    }
}
let newclass=NewClass(i:200000,j:10,k:"Hello") //实例化对象
println("value1=\(newclass.value1)")
println("value2=\(newclass.value2)")
println("value3=\(newclass.value3)")
```

此时运行程序,会看到如下的结果:

```
value1=200000
```

```
value2=10.0
value3=Hello
```

(2) 便利构造器

便利构造器就没有指定构造器那么重要了。开发者可以定义便利构造器来调用同一个类中的指定构造器，并为其参数提供默认值，也可以定义便利构造器来创建一个特殊用途或特定输入的实例。便利构造器需要使用 `convenience` 关键字和 `init` 关键字进行定义，其语法形式如下：

```
convenience init(参数名1.数据类型, 参数名2.数据类型, ...) {
    ...
}
```

便利构造器的调用形式如下：

```
let/var 对象名=类名(参数名1:内容, 参数名2:内容, ...)
```

【示例 4-41】 以下将实现便利构造器的定义和声明。代码如下：

```
class NewClass{
    var age: Int
    var name:String
    init(name: String, age: Int) {
        self.age = age
        self.name=name
    }
    //定义具有一个参数的便利构造器
    convenience init(name: String) {
        self.init(name: name, age: 15)
    }
}
let newclass1=NewClass(name:"Tom",age:25)
println("name=\(newclass1.name)")
println("age=\(newclass1.age)")
let newclass2=NewClass(name:"Dive") //调用便利构造器
println("name=\(newclass2.name)")
println("age=\(newclass2.age)")
```

此时运行程序，会看到如下的结果：

```
name=Tom
age=25
name=Dive
age=15
```

在使用便利构造器时，需要注意以下两点：

- ☐ 在一个便利构造器中必须要调用一个指定构造器。
- ☐ 便利构造器不可以和指定构造器声明个数相同、顺序相同的参数。

3. 构造器代理

类的构造器代理实现规则和形式是非常麻烦的。在此处，我们主要讲解在基类（基类就是不继承任何的类）中的构造器代理。这是在类中比较简单的构造器代理。对于基类中的构造器代码其实在自定义构造器中已经存在了，在定义便利构造器时就使用到了构造器代理。如以下的代码：


```
class NewClass{
    var age: Int
    var name:String
    init(name: String, age: Int) {
        self.age = age
        self.name=name
    }
    //定义具有一个参数的便利构造器
    convenience init(name: String) {
        self.init(name: name, age: 15) //在便利构造器中实现对指定构造器的调用
    }
}
```

此代码是【示例 4-41】中的代码片段，在代码中加粗的部分就是构造器代码。

4. 构造器的继承和重写

类的构造器代理实现规则和形式是非常复杂的，因为类可以实现继承。以下是构造器之间的代理调用使用规则。

- ❑ 指定构造器（Designated）：指定构造器必须要调用其直接父类中的指定构造器。
- ❑ 便利构造器（Convenience）：便利构造器必须调用同一类中定义的其他构造器。
- ❑ 便利构造器必须最终调用指定构造器：便利构造器必须最终以调用一个指定构造器结束。这些规则可以使用图 4.3 来进行说明。

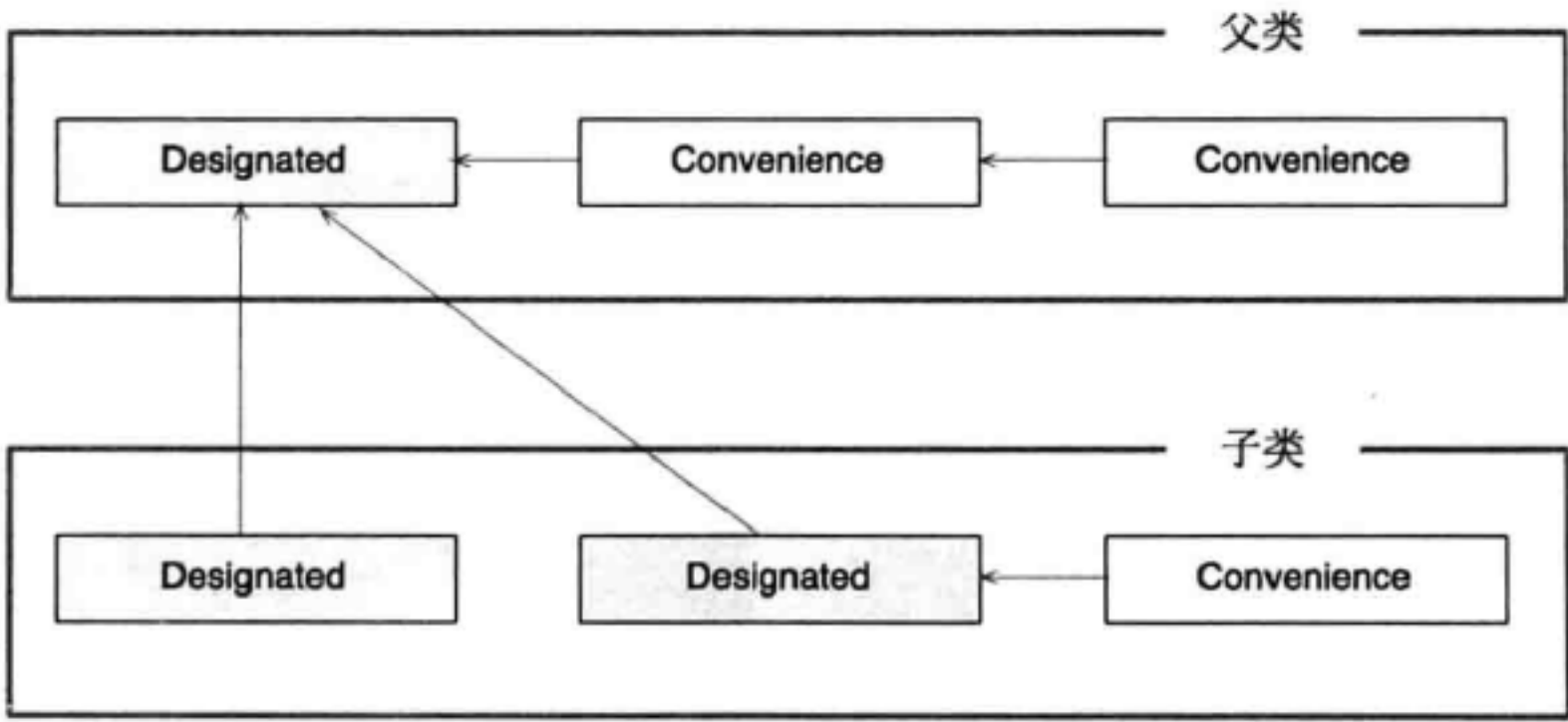


图 4.3 指定构造器和便利构造器指定的规则 1

如图 4.3 所示，父类中包含一个指定构造器和两个便利构造器。其中一个便利构造器调用了另外一个便利构造器，而后者又调用了唯一的指定构造器。这满足了上面提到的规则 2 和规则 3。这个父类没有自己的父类，所以规则 1 没有用到。

子类中包含两个指定构造器和一个便利构造器。便利构造器必须调用两个指定构造器中的任意一个，因为它只能调用同一个类里的其他构造器。这满足了上面提到的规则 2 和规则 3。而两个指定构造器必须调用父类中唯一的指定构造器，这满足了规则 1。如下展示了一种针对 4 个类的更复杂的类层级结构。它演示了指定构造器是如何在类层级中充当“管道”作用的，在类的构造器链上简化了类之间的相互关系，如图 4.4 所示。

(1) 构造器的继承

Swift 中的子类默认不会继承父类的构造器，以防止父类的简单构造被子类继承，并错误地创建子类的实例。如果希望子类能继承父类相同的构造器，需要定制子类的构造器。

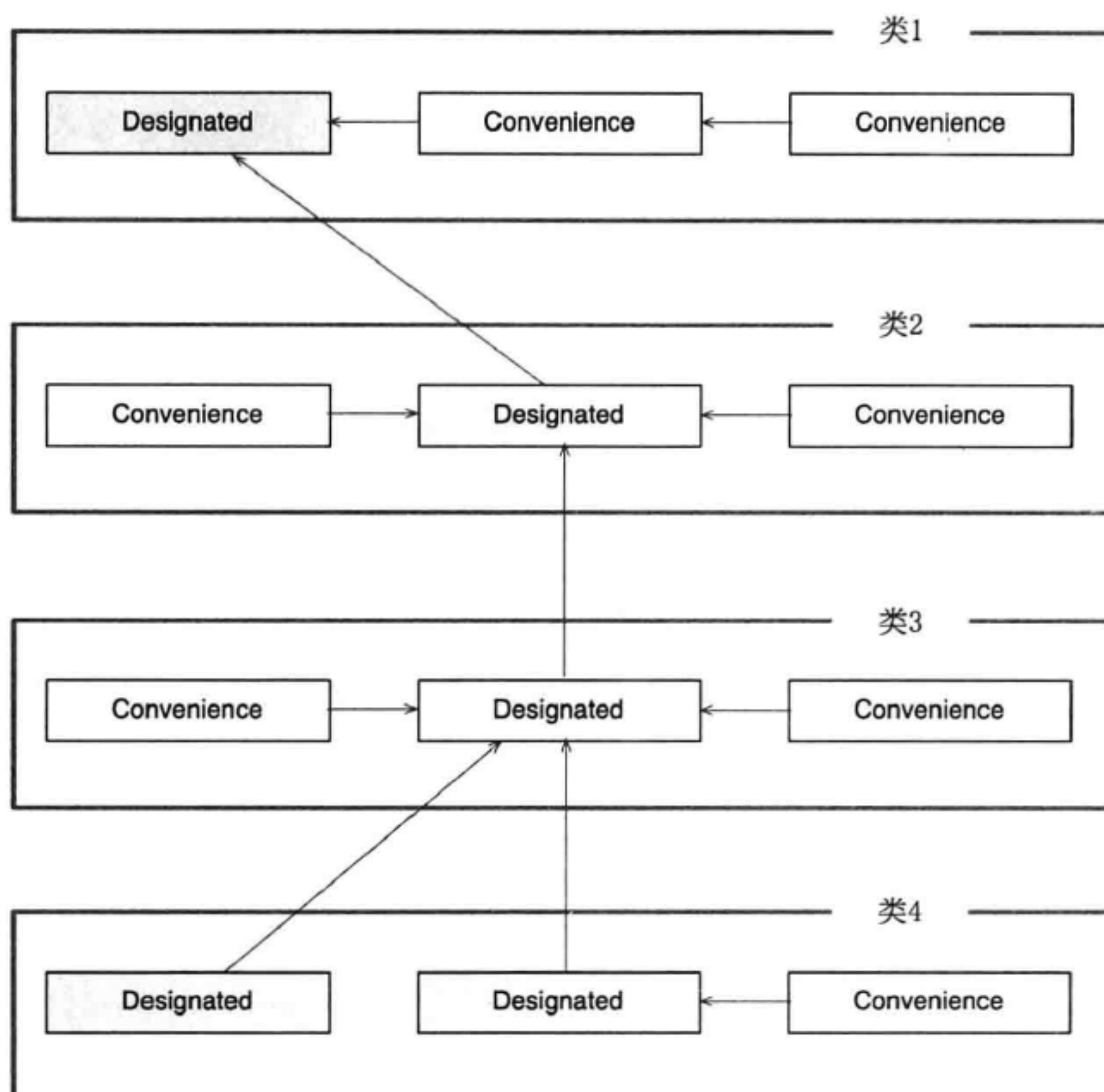


图 4.4 指定构造器和便利构造器指定的规则 2

【示例 4-42】 以下让子类 NewClass2 继承父类 NewClass1 中的构造器。代码如下：

```
class NewClass1{
    var value1:Int=0
    var value2:String=""
    //指定构造器
    init(val1:Int,val2:String) {
        value1=val1
        value2=val2
    }
    //便利构造器
    convenience init(){
        self.init(val1:100,val2:"无内容")
    }
}
//子类 NewClass1
class NewClass2:NewClass1{
}
let newclass1=NewClass2(val1: 5, val2: "Hello")           //继承指定构造器
println(newclass1.value1)
println(newclass1.value2)
let newclass2=NewClass2()                                 //继承便利构造器
println(newclass2.value1)
println(newclass2.value2)
```

此时运行程序，会看到如下的结果：

```
Hello
100
无内容
```

(2) 构造器的重写


构造器进行重写时，指定构造器需要添加 `override` 关键字。

【示例 4-43】 以下将实现指定构造器的重写。代码如下：

```
class NewClass1{
    var value1:Int=0
    var value2:Int=0
    init(val1:Int,val2:Int) {
        value1=val1
        value2=val2
    }
}
class NewClass2:NewClass1{
    var value=0
    //重载指定构造器
    override init(val1:Int,val2:Int) {
        super.init(val1:200,val2:600)
    }
}
let newclass1=NewClass2(val1: 8, val2: 2)
println(newclass1.value1)
println(newclass1.value2)
```

此时运行程序，会看到如下的结果：

```
200
600
```

 **注意：** 便利构造器在实现重写时，不需要添加 `override` 关键字。

4.5.3 析构方法

析构方法使用在类中，当一个类的实例释放之前，析构方法被立即调用，用来执行某些特定的清除工作。本小节将讲解析构方法的定义和使用。

1. 定义析构方法

析构方法使用关键字 `deinit` 进行定义，其语法形式如下：

```
deinit {
    ...
}
```

2. 使用析构方法

指定了析构方法的定义后，就可以在类中使用了。

【示例 4-44】 以下将在类中对析构方法进行使用。代码如下：

```
class NewClass{
    deinit{
        println("析构方法")
    }
}
```

```

    }
}
var newclass:NewClass?=NewClass()
newclass=nil

```

此时运行程序，会看到如下的结果：

析构方法

在使用析构方法时，需要注意以下 3 点。

- (1) 在一个类中只可以定义一个析构方法。
- (2) 析构方法的调用是隐式调用，不可以是显式调用（对象去调用析构方法）。
- (3) 析构方法只可以使用在类中，不可以使用在结构或者枚举等类型中。

4.6 扩展和协议

扩展和协议在很多面向对象的编程语言中存在，当然 Swift 也不例外。使用扩展和协议，可以在面向对象的编程语言中对已有的类型进行扩展和修改。

4.6.1 扩展

扩展就是向一个已经存在的类以及数值类型添加新的功能，如属性、方法和下标脚本等。本节将主要讲解扩展的定义，以及方法、属性、构造器、下标脚本的扩展等内容。

1. 定义扩展

扩展是使用关键字 `extension` 进行定义的，其语法形式如下：

```

extension 类型 {
    ...
}

```

2. 扩展属性

扩展可以向已有的类型（类、数值类型）中添加计算属性和类型属性。

【示例 4-45】 以下将创建一个空类 `NewClass`，然后使用扩展，向空类中添加计算属性和类型属性。代码如下：

```

class NewClass{
}
extension NewClass{
    //添加计算属性
    var value1:String{
        return "Hello,Swift"
    }
    //添加类型属性
    class var value2:Double{
        return 1000
    }
}

```



```

}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var newclass=NewClass()
        println("value1=\(newclass.value1)")           //访问计算属性
        println("value2=\(NewClass.value2)")           //访问类型属性
    }
}

```

此时运行程序，会看到如下的结果：

```

value1=Hello,Swift
value2=1000.0

```

3. 扩展构造器

扩展可以向已有的类型中添加构造器。

【示例 4-46】 以下将创建一个结构 NewStruct，然后使用扩展，向此结构中添加构造器。代码如下：

```

struct NewStruct {
    var red:Double
    var green:Double
    var blue:Double
}
extension NewStruct{
    //添加构造器
    init(Red: Double, Green: Double, Blue: Double) {
        red = Red
        green = Green
        blue = Blue
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let magenta = NewStruct(Red: 1.0, Green: 0.0, Blue: 1.0)
        println("red=\(magenta.red)")
        println("green=\(magenta.green)")
        println("blue=\(magenta.blue)")
    }
}

```

此时运行程序，会看到如下结果：

```

red=1.0
green=0.0
blue=1.0

```

4. 扩展方法

扩展也可以向已有的类型中添加方法。

【示例 4-47】 以下将创建一个空类 NewClass，然后使用扩展，向空类中添加实例方法和类型方法。代码如下：

```

class NewClass{
}

```

```

extension NewClass{
    //实例方法
    func printStr(value:String){
        for i in value {
            println(i)
        }
    }
    //类型方法
    class func add(value1:Int,value2:Int)->Int{
        return value1+value2
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var newclass=NewClass()
        newclass.printStr("Hello")
        println(NewClass.add(50, value2: 100))
    }
}

```

此时运行程序，会看到如下的结果：

```

H
e
l
l
o
150

```

5. 扩展下标脚本

扩展也可以向已有的类型中添加下标脚本。

【示例 4-48】 以下将创建一个空类 NewClass，然后使用扩展，向空类中添加下标脚本。代码如下：

```

class NewClass{
}
extension NewClass{
    //添加下标脚本
    subscript(index: Int,value:Int) -> Int {
        return index+value
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var newclass=NewClass()
        println(newclass[2,8])
    }
}

```

此时运行程序，会看到如下的结果：

```

10

```

4.6.2 协议

协议用于声明完成某项任务或功能所必需的方法和属性。这些方法和属性并不在协议中具体实现，而只用来描述这些实现应该是什么样的。对于在协议中声明的方法和属性需要在协议实现的类型（遵循者）中进行具体实现。本小节将讲解协议的定义、实现，以及协议的成员声明等内容。

1. 定义协议

协议的定义需要使用 `protocol` 关键字，其语法形式如下：

```
protocol 协议名称 {
    ...
}
```

2. 实现协议

协议在定义好以后，就可以在类型中实现了。在数值类型中协议的实现都是一样的，以结构为例，协议在结构中的实现形式如下：

```
struct 结构名称:协议名 {
    ...
}
```

同样的方式适用于其他数值类型。对于一个具有指定类型的枚举类型来说，在冒号后面首先跟着的是枚举类型指定的数据类型，然后才是协议名称，其实现形式如下：

```
enum 枚举类型名:数据类型, 协议名 {
    ...
}
```

其中，数据类型和协议名之间使用逗号分隔。


对于类来说，协议在类中的实现形式和在数值类型中的实现形式是一样的，其语法形式如下：

```
class 类名:协议名 {
    ...
}
```

如果一个类含有父类的同时也采用了协议，应当把父类放在所有的协议之前，其形式如下：

```
class 类名:父类名, 协议名 {
    ...
}
```

其中，父类名和协议名中间同样使用逗号分隔。

 **注意：**在类型中，可以对多个协议进行实现，实现多个协议时，各协议之间用逗号分隔。以类为例，其语法形式如下：


```
class 类名:协议名1,协议名2,协议名3,...{
    ...
}
```

同样的方式适用于其他类型。

3. 协议的成员声明——属性

在协议中可以声明属性。它可以为其遵循者（即实现协议的类型）提供特定名称与类型的实例属性或类型属性，而不管其是存储型属性还是计算型属性。以下将主要讲解这两种属性的声明。

(1) 实例属性

实例属性其实就是计算型属性和存储型属性。其声明形式如下：

```
protocol 协议名称{
    var 属性名称:数据类型 {get}                //声明一个只读的实例属性
    var 属性名称:数据类型 {get set}             //声明一个可读写的实例属性
}
```

(2) 类型属性

类型属性的声明和实例属性的声明一般相同，只不过需要添加一个代表类型的关键字，其中，类的类型属性定义形式如下：

```
protocol 协议名{
    class var 属性名1: 数据类型 { get }          //声明一个只读的类型属性
    class var 属性名2: 数据类型 { get set }       //声明一个可读写的类型属性
}
```

数值类型的类型属性定义形式如下：

```
protocol 协议名{
    static var 属性名1: 数据类型 { get }         //声明一个只读的类型属性
    static var 属性名2: 数据类型 { get set }      //声明一个可读写的类型属性
}
```

【示例 4-49】 以下将在协议 NameProtocol 中声明一个只读的实例属性和一个只读的类型属性，然后在遵循者中 NewClass 类中进行实现。代码如下：

```
protocol NameProtocol{
    var name: String { get }                    //声明只读的属性 name
    class var age:Int{get}
}
class NewClass:NameProtocol{
    //定义协议中只读的实例属性 name
    var name: String{
        return "Hello"
    }
    //定义协议中只读的类型属性 age
    class var age:Int{
        return 20
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
```

```

        let newclass=NewClass()
        println(newclass.name)
        println(NewClass.age)
    }
}

```

此时运行程序，会看到如下的结果：

```

Hello
20

```

4. 协议的成员声明——方法

协议可以要求指定实例方法和类型方法被一致的类型实现。这些方法被写为协议定义的一部分，跟普通实例和类型方法完全一样，但是没有大括号或方法体。以下就是在协议中这两种方法声明的形式：

(1) 实例方法

实例方法的声明形式如下：

```

protocol 协议名 {
    func 方法名(参数名 1.数据类型, 参数名 2.属性类型, ...) -> 返回值的数据类型
}

```

实例方法在协议中声明好以后，需要在遵循者中进行实现。

(2) 类型方法

类的类型方法的声明形式如下：

```

protocol 协议名 {
    class func 方法名(参数名 1.数据类型, 参数名 2.属性类型, ...) -> 返回值的数据类型
}

```

数值类型的类型方法声明形式如下：

```

protocol 协议名 {
    static func 方法名(参数名 1.数据类型, 参数名 2.属性类型, ...) -> 返回值的数据类型
}

```

类型方法在协议中声明好以后，需要在遵循者中进行实现。

【示例 4-50】 以下将在协议 NewProtocol 中声明一个实例方法和一个类型方法，然后在遵循者中 NewClass 类中进行实现。代码如下：

```

protocol NewProtocol{
    func random() -> Double //声明实例方法
    class func printNumber(i:Int) //声明类型方法
}
class NewClass:NewProtocol{
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    //实例方法，实现随机数的生成
    func random() -> Double {
        lastRandom = ((lastRandom * a + c) % m)
        return lastRandom / m
    }
}

```



```

//类型方法，实现数字的输出
class func printNumber(i:Int){
    println(i)
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let newclass=NewClass()
        println(newclass.random())
        NewClass.printNumber(5)
    }
}

```

此时运行程序，会看到如下的结果：

```

0.37464991998171
5

```

5. 协议的成员声明——可变方法

迫于某种需求，有时不得不在方法中更改实例的所属类型，我们称这样的方法为可变方法。可变方法的定义就是在方法的前面添加 mutating 关键字。它表示可以在该方法中修改实例及其属性的所属类型，其语法形式如下：

```

protocol 协议名 {
    mutating func 方法名(参数名1:数据类型, 参数名2:数据类型, ...)
}

```

【示例 4-51】 以下将在协议中定义可变方法，然后在遵循者中实现此方法，实现开关的功能。代码如下：

```

protocol Togglable {
    mutating func toggle() //声明可变方法
}
enum OnOffSwitch:Int,Togglable {
    case Off=0
    case On=1
    //定义可变方法
    mutating func toggle() {
        switch self {
            case .Off:
                self = On
                println(self.rawValue)
            case .On:
                self = Off
                println(self.rawValue)
        }
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var lightSwitch = OnOffSwitch.On
        println(OnOffSwitch.On.rawValue)
        lightSwitch.toggle()
    }
}

```


此时运行程序，会看到如下的结果：

```
1
0
```

4.6.3 可选协议

可选协议可以使在协议中声明的成员在遵循者中不必都实现，这样可以使代码量减少，从而解决了代码冗余的问题。

1. 定义可选协议

可选协议的定义需要使用@objc 关键字，其定义形式如下：

```
@objc protocol 协议名 {
    ...
}
```

2. 声明可选成员

开发者可以在可选协议中声明可选成员。声明可选成员其实很简单，就是在声明的属性和方法的前方加上关键字 optional。以声明一个可选的只读属性为例，它的声明形式如下：

```
@objc protocol 协议名 {
    optional var 属性名:数据类型{get}
}
```

同样的方式适用于其他协议成员。可选成员可以让遵循者选择是否实现这些成员。

【示例 4-52】 以下将在可选类型中声明可选协议。代码如下：

```
@objc protocol CounterProtocol {
    //声明可选成员
    optional func printlnCount(count: Int) -> Int
    optional var value: Int { get }
}
//定义协议的遵循者 NewClass 类
class NewClass:CounterProtocol{
    var value:Int{
        return 500
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var newclass=NewClass()
        println(newclass.value)
    }
}
```

此时运行程序，会看到如下的结果：

```
500
```

4.6.4 使用协议类型

尽管协议本身并不实现任何功能，但是协议可以被当做类型来使用。本小节将讲解使用协议类型可以用来做什么。

1. 协议类型作为常量、变量等的数据类型

协议类型可以作为常量、变量以及属性的数据类型。

【示例 4-53】 以下将协议类型用在变量的数据类型。代码如下：

```
protocol Name {
    var name: String{get}
}
//定义类 NewClass, 并实现 Name 协议
class NewClass:Name{
    var name:String="Swift"
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var value:Name=NewClass()
        println(value.name)
    }
}
```

此时运行程序，会看到如下的结果：

Swift

2. 协议类型的返回值或参数

协议类型可以作为函数、方法、构造器中的参数类型或返回值类型。

【示例 4-54】 以下将协议类型作为构造器的参数类型。代码如下：

```
protocol Name {
    var name: String{get}
}
//定义类 NewClass, 并实现 Name 协议
class NewClass:Name{
    var name:String{
        return "Hello"
    }
}
//定义类 NewClass1
class NewClass1{
    var value:Name
    //将协议类型作为构造器的参数
    init(value:Name) {
        self.value = value
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var newclass=NewClass1(value:NewClass())
    }
}
```



```
        println(newclass.value.name)
    }
}
```

此时运行程序，会看到如下的结果：

```
Hello
```

3. 协议类型作为集合的元素类型

协议类型也可以作为数组、字典或其他容器中的元素类型。

【示例 4-55】 以下将用协议类型作为数组的元素类型。代码如下：

```
protocol Name {
    var name: String {get}
}
//定义类 NewClass1，并实现 Name 协议
class NewClass1:Name{
    var name:String{
        return "Swift"
    }
}
//定义类 NewClass2，并实现 Name 协议
class NewClass2:Name{
    var name:String{
        return "Hello"
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let array:[Name]=[NewClass1(),NewClass2(),NewClass1()]
        var i=0
        //遍历
        for i=0;i<3;++i {
            println("array[\(i)]=\(array[i].name)")
        }
    }
}
```


此时运行程序，会看到如下的结果：

```
array[0]=Swift
array[1]=Hello
array[2]=Swift
```

4.6.5 协议的继承

协议和类一样，也是可以实现继承的。它可以继续一个或者多个协议，其语法形式如下：

```
protocol 协议名 n: 协议名 1, 协议名 2, ... {
    ... //内容
}
```

 **注意：** 多个协议需要使用逗号分隔。

【示例 4-56】 以下将实现协议的继承,即让协议 NewProtocol2 继承协议 NewProtocol1。代码如下:

```
protocol NewProtocol1{
    func printHello()
}
//协议 NewProtocol2 继承 NewProtocol1
protocol NewProtocol2:NewProtocol1{
    func printSwift()
}
class NewClass:NewProtocol2 {
    func printHello(){
        println("Hello")
    }
    func printSwift() {
        println("Swift")
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let newclass=NewClass()
        newclass.printHello()
        newclass.printSwift()
    }
}
```

此时运行程序, 会看到如下的结果:

```
Hello
Swift
```

4.6.6 协议的组合

一个协议可以采用 protocol<协议名 1,协议名 2,...>这样的形式将多个协议组合在一起,这样的协议称为协议组合。

【示例 4-57】 以下将使用协议组合实现对生日祝福的输出。代码如下:

```
protocol NamedProtocol {
    var name: String { get }
}
//定义协议 AgedProtocol
protocol AgedProtocol {
    var age: Int { get }
}
//定义结构 Person
struct Person: NamedProtocol, AgedProtocol {
    var name: String
    var age: Int
}
//定义函数
func wishHappyBirthday(celebrator: protocol<NamedProtocol, AgedProtocol>) {
    println(" \ (celebrator.name) 生日快乐 - 你\ (celebrator.age) 岁了!")
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

```

        let birthdayPerson = Person(name: "Tom", age: 100)
        wishHappyBirthday(birthdayPerson)
    }
}

```

此时运行程序，会看到如下的结果：

```
Tom 生日快乐 - 你 100 岁了!
```

4.6.7 检查协议的一致性

开发者可以使用类型检测中描述的 `is` 和 `as` 运算符检测器检查协议的一致性，或转化协议类型。对于检查类型和转换类型在前面的章节中已经讲解过了。在协议类型中，`is` 和 `as` 的功能如下：

- ❑ `is` 操作符用来检查实例是否遵循了某个协议。
- ❑ `as?` 返回一个可选值，当实例遵循协议时，返回该协议类型；否则返回 `nil`。
- ❑ `as` 用以强制向下转换。

【示例 4-58】 以下使用 `as?` 来判断类实例是否遵循了 `HasArea` 协议。代码如下：

```

//协议 HasArea
@objc protocol HasArea {
    var area: Double { get }
}
//类 Circle, 遵守协议 HasArea
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    //可读的计算属性
    var area: Double {
        return pi * radius * radius
    }
    //定义指定构造器
    init(radius: Double) {
        self.radius = radius
    }
}
//类 Country, 遵守协议 HasArea
class Country: HasArea {
    var area: Double
    //定义指定构造器
    init(area: Double) {
        self.area = area
    }
}
//类 Animal
class Animal {
    var legs: Int
    //定义指定构造器
    init(legs: Int) {
        self.legs = legs
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
    }
}

```



```

let objects = [Circle(radius: 2.0),Country(area: 243610),Animal(legs: 4)]
//遍历数组
let aa:[AnyObject] = [Circle(radius: 2.0),Country(area: 243610),
Animal(legs: 4)]
for item:AnyObject in aa {
    //判断是否遵守协议 HasArea
    if let objectWithArea = item as? HasArea {
        println(objectWithArea.area)
    }else{
        println("没有遵守协议 HasArea")
    }
}
}
}

```

此时运行程序，会看到如下的结果：

```

12.5663708
243610.0
没有遵守协议 HasArea

```

4.6.8 委托

委托是一个简单的遵守协议的变量，是一个典型的用来通知事件或执行各种子任务的类。

【示例 4-59】 以下将使用委托实现模拟人说话的功能。代码如下：

```

//定义可选协议
@objc protocol Speaker {
    func Speak()
    optional func TellJoke()
}
//定义协议
protocol DateSimulatorDelegate {
    func dateSimulatorDidStart(sim:DateSimulator, a:Speaker, b:Speaker)
    func dateSimulatorDidEnd(sim:DateSimulator, a: Speaker, b:Speaker)
}
//定义 LoggingDateSimulator 类
class LoggingDateSimulator:DateSimulatorDelegate {
    //输出"Date started!"字符串
    func dateSimulatorDidStart(sim:DateSimulator, a:Speaker, b:Speaker) {
        println("Date started!")
    }
    //输出"Date ended!"字符串
    func dateSimulatorDidEnd(sim:DateSimulator, a: Speaker, b: Speaker) {
        println("Date ended!")
    }
}
//定义 Vicki 类
class Vicki: Speaker {
    //输出"Hello, I am Vicki!" 字符串
    func Speak() {
        println("Hello, I am Vicki!")
    }
    //输出"Q: What did Sushi A say to Sushi B?"字符串
    func TellJoke() {

```



```

        println("Q: What did Sushi A say to Sushi B?")
    }
}
class Ray: Speaker {
    //输出"Yo, I am Ray!"字符串
    func Speak() {
        println("Yo, I am Ray!")
    }
    //输出"Q: Whats the object-oriented way to become wealthy?"字符串
    func TellJoke() {
        println("Q: Whats the object-oriented way to become wealthy?")
    }
}
class DateSimulator {
    let a:Speaker
    let b:Speaker
    var delegate:DateSimulatorDelegate? //该属性用来获取一个遵守这个协议的类
    //构造器
    init(a:Speaker, b:Speaker) {
        self.a = a
        self.b = b
    }
    func simulate() {
        delegate?.dateSimulatorDidStart(self, a:a, b: b)
        //输出"Date started!"字符串
        println("Off to dinner...")
        a.Speak()
        b.Speak()
        println("Walking back home...")
        a.TellJoke?()
        b.TellJoke?()
        delegate?.dateSimulatorDidEnd(self, a:a, b:b)
        //输出"Date ended!"字符串
    }
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let sim = DateSimulator(a:Vicki(), b:Ray())
        sim.simulate()
    }
}

```

此时运行程序，会看到如下的结果：

```

Off to dinner...
Hello, I am Vicki!
Yo, I am Ray!
Walking back home...
Q: What did Sushi A say to Sushi B?
Q: Whats the object-oriented way to become wealthy?

```

4.7 运算符重载

运算符重载其实是一种特殊的函数。它让已有的运算符也可以对自定义的类和结构进行运算。通过运算符重载可以扩展运算符在类或者结构中的作用。本节将主要讲解关于运

算符重载的一些内容。

4.7.1 算术运算符重载

算术运算符分为+、-、*、/、%这 5 种，它们都属于中置运算符。它的重载语法形式如下：

```
func 算术运算符 (参数名 1:数据类型, 参数名 2:数据类型) -> 返回值的数据类型 {
    ...
    return 返回数据
}
```

【示例 4-60】 以下将定义一个加法运算符的重载，使它可以在结构中做加法运算。代码如下：

```
//定义结构
struct Vector2D {
    var x = 0.0
    var y = 0.0
}
//实现加法运算符的重载
func + (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y + right.y)
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let vector = Vector2D(x: 3.0, y: 1.0)
        let anotherVector = Vector2D(x: 8.0, y: 6.0)
        let combinedVector = vector + anotherVector //实现加法运算
        println(combinedVector.x)
        println(combinedVector.y)
    }
}
```

此时运行程序，会看到如下的结果：

```
11.0
7.0
```

4.7.2 前置运算符和后置运算符重载

在操作数之前的运算符就是前置运算符，如-a；在操作数之后的运算符就是后置运算符，如 i++。实现一个前置运算符或者后置运算符重载时，需要在定义该运算符时在关键字 func 之前标注 prefix 或 postfix 属性。定义前置运算符重载的语法形式如下：

```
prefix func 运算符 (参数名:数据类型) ->数据类型{
    return 返回数据
}
```

定义后置运算符重载的语法形式如下：

```
postfix func 运算符 (参数名:数据类型) ->数据类型{
```



```
    return 返回数据
}
```

【示例 4-61】 以下将实现-前置运算符的重载，使它可以对结构中的属性值变为负数。代码如下：

```
//定义结构
struct NewStruct{
    var x=0
    var y=0
}
//运算符的重载
prefix func -(vector:NewStruct)->NewStruct{
    return NewStruct(x:-vector.x,y:-vector.y)
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        let positive=NewStruct(x:20,y:100) //实例化对象
        let negative = -positive //实现一元减运算符的功能，即正数变负，负数变正
        println(negative.x)
        println(negative.y)
        let alsoPositive = -negative//实现一元减运算符的功能，即正数变负，负数变正
        println(alsoPositive.x)
        println(alsoPositive.y)
    }
}
```

此时运行程序，会看到如下的结果：

```
-20
-100
20
100
```

4.7.3 复合运算符重载

复合赋值运算符同样也是可以重载的，其语法形式如下：

```
func 复合运算符 (inout 参数名 1:数据类型, 参数名 2:数据类型) {
    ...
}
```

其中需要将运算符的左参数设置成 inout，因为这个参数会在运算符重载函数内直接修改它的值。

【示例 4-62】 以下将定义一个加法复合赋值运算符+=的重载，实现结构的加法运算以及赋值。代码如下：

```
//定义结构
struct Vector2D {
    var x = 0.0
    var y = 0.0
}
//加法运算符的重载
func + (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y + right.y)
}
```



```
//复合赋值运算符的重载
func += (inout left: Vector2D, right: Vector2D) {
    left = left + right
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var original = Vector2D(x: 1.0, y: 2.0)
        let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
        original += vectorToAdd //实现两个结构的相加，并赋值
        //输出
        println(original.x)
        println(original.y)
    }
}
```

此时运行程序，会看到如下的结果：

```
4.0
6.0
```

4.7.4 比较运算符重载

比较运算符进行重载的语法形式如下：

```
func 比较运算符 (参数名 1:数据类型, 参数名 2: 数据类型) -> 数据类型{
    return 返回的数据
}
```

【示例 4-63】 以下将重载运算符==，让其实现判断两个结构是否相等的功能。代码如下：

```
//定义结构
struct Vector2D {
    var x = 0.0
    var y = 0.0
}
//定义==运算符的重载
func == (left: Vector2D, right: Vector2D) -> Bool {
    return (left.x == right.x) && (left.y == right.y)
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let one = Vector2D(x: 2.0, y: 3.0)
        let two = Vector2D(x: 2.0, y: 3.0)
        let three=Vector2D(x: 3.0, y: 3.0)
        //实现判断 one 和 two 对象是否相等
        if one==two{
            println("比较的两个结构相等")
        }else{
            println("比较的两个结构不相等")
        }
        //实现判断 one 和 three 对象是否相等
        if one==three{
            println("比较的两个结构相等")
        }else{
```

```
        println("比较的两个结构不相等")
    }
}
}
```

此时运行程序，会看到如下的结果：

```
比较的两个结构相等
比较的两个结构不相等
```

4.7.5 自定义运算符

自定义运算符是由开发者定义的。它分为前置自定义运算符、中置自定义运算符，以及后置自定义运算符这3种。以下就是对这3种自定义运算符的介绍。

1. 前置自定义运算符的重载

前置自定义运算符的定义形式如下：

```
prefix operator 自定义运算符 {...}
```

重载的形式如下：

```
prefix func 自定义运算符(inout 参数名: 数据类型) -> 数据类型 {
    ...
    return 返回数据
}
```

其中，operator 关键字用来定义自定义运算符，prefix 关键字表示此运算符是前置运算符。

【示例 4-64】 以下将定义一个***自定义运算符，然后实现它的重载，即实现定义求取结构的属性值的平方。代码如下：

```
//定义结构
struct Vector2D {
    var x = 0.0
    var y = 0.0
}

prefix operator *** {} //定义前置自定义运算符
//定义乘法运算符的重载
func * (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x * right.x, y: left.y * right.y)
}
//定义乘法复合赋值运算符的重载
func *= (inout left: Vector2D, right: Vector2D) {
    left = left * right
}
//定义自定义运算符的重载
prefix func *** (inout vector: Vector2D) -> Vector2D {
    vector *= vector
    return vector
}
class ViewController: UIViewController {
    override func viewDidLoad() {
```



```

    super.viewDidLoad()
    var toBeDoubled = Vector2D(x: 8.0, y: 5.0)
    //实现自定义运算符的运算
    ***toBeDoubled
    println(toBeDoubled.x)
    println(toBeDoubled.y)
    //实现自定义运算符的运算
    ***toBeDoubled
    println(toBeDoubled.x)
    println(toBeDoubled.y)
}
}

```

此时运行程序，会看到如下的结果：

```

64.0
25.0
4096.0
625.0

```

2. 中置自定义运算符的重载

中置自定义运算符的定义形式如下：

infix operator 自定义运算符 {...}

重载的形式如下：

```

func 自定义运算符 (参数名 1: 数据类型, 参数名 2: 数据类型) -> 数据类型 {
    ...
    return 返回数据
}

```

【示例 4-65】 以下将定义一个+-自定义运算符，然后实现它的重载，即实现结构中第一个属性值的相加，第二个属性值的相减。代码如下：

```

//定义结构
struct Vector2D {
    var x = 0.0
    var y = 0.0
}
infix operator +- {} //定义中置自定义运算符
//定义自定义运算符的重载
func +- (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y - right.y)
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        //实例化对象
        let firstVector = Vector2D(x: 1.0, y: 2.0)
        let secondVector = Vector2D(x: 3.0, y: 4.0)
        let plusMinusVector = firstVector +- secondVector
        println(plusMinusVector.x)
        println(plusMinusVector.y)
    }
}

```

此时运行程序，会看到如下的结果：


```
4.0
-2.0
```

3. 后置自定义运算符的重载

后置自定义运算符的定义形式如下：

```
postfix operator 自定义运算符 {…}
```

重载的形式如下：

```
postfix func 自定义运算符(inout 参数名: 数据类型) -> 数据类型 {
    ...
    return 返回数据
}
```

其中，postfix 表示定义的自定义运算符是后置运算符。

【示例 4-66】 以下将定义一个----自定义运算符，然后实现它的重载，即实现结构中属性值的自减 10。代码如下：

```
//定义结构
struct Vector2D {
    var x = 0.0
    var y = 0.0
}
postfix operator ---- {} //定义后置自定义运算符
//定义加法运算符的重载
func - (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x - right.x, y: left.y - right.y)
}
//定义加法复合赋值运算符的重载
func -= (inout left: Vector2D, right: Vector2D) {
    left = left - right
}
//定义自定义运算符的重载
postfix func ---- (inout vector: Vector2D) -> Vector2D {
    vector -= Vector2D(x: 10.0, y: 10.0)
    return vector
}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var toIncrement = Vector2D(x: 3.0, y: 4.0)
        //实现自定义运算符的功能
        toIncrement----
        println(toIncrement.x)
        println(toIncrement.y)
        //实现自定义运算符的功能
        toIncrement----
        println(toIncrement.x)
        println(toIncrement.y)
    }
}
```

此时运行程序，会看到如下的结果：

```
-7.0
-6.0
```

```
-17.0
-16.0
```

4.8 泛型

泛型是 Swift 引入的一个新的特性。使用它可以确保开发者写出灵活的、可重用的函数或定义出任何你所确定好需求的类型。这样，就可以提供代码的重用性，并且避免了代码的重复。本节将讲解一些关于泛型的内容。

4.8.1 泛型函数

泛型函数可以工作于任何类型，其定义形式如下：

```
func 函数名<T>(参数名 1:T, 参数名 2:T, 参数名 3:T, ...) -> 返回值类型 {
    ...
    return 返回值
}
```

其中，T 表示一个占位类型名（类型参数）。当然，T 也可以被换为其他的字符或字符串。

【示例 4-67】 以下将实现任意类型的两个数的交换。代码如下：

```
//定义泛型函数 swapTwoValue
func swapTwoValue<T>(inout a: T, inout b: T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
var someInt = 3
var anotherInt = 107
println("交换前: someInt = \(someInt), anotherInt = \(anotherInt)")
swapTwoValue(&someInt, &anotherInt) //实现两个整数的交换
println("交换后: someInt = \(someInt), anotherInt = \(anotherInt)")
var someString = "Swift"
var anotherString = "iOS 8"
println("交换前: someString = \(someString), anotherString = \(anotherString)")
swapTwoValue(&someString, &anotherString) //实现两种字符串的交换
println("交换后: someString = \(someString), anotherString = \(anotherString)")
var someDouble = 13.5555
var anotherDouble = 65.5555
println("交换前: someDouble = \(someDouble), anotherDouble = \(anotherDouble)")
swapTwoValue(&someDouble, &anotherDouble) //实现两个双精度数据的交换
println("交换后: someDouble = \(someDouble), anotherDouble = \(anotherDouble)")
```

此时运行程序，会看到如下的结果：

```
交换前: someInt = 3, anotherInt = 107
交换后: someInt = 107, anotherInt = 3
交换前: someString = Swift, anotherString = iOS 8
交换后: someString = iOS 8, anotherString = Swift
交换前: someDouble = 13.5555, anotherDouble = 65.5555
```


交换后: someDouble = 65.5555, anotherDouble = 13.5555

4.8.2 泛型类型

通常在泛型函数中, Swift 允许开发者定义自己的泛型类型。这些泛型类型包括自定义类、结构体和枚举等, 它们可以作用于任何类型。以下将主要讲解这 3 种泛型类型。

1. 泛型枚举

泛型枚举的定义形式如下:

```
enum 枚举名称<T>{
    ...
}
```

其中, 需要使用尖括号<>定义泛型枚举。尖括号中的占位类型名也可以是其他。

【示例 4-68】 以下将输出任意类型的内容。代码如下:

```
enum NewEnum<T>{
    //定义类型方法
    static func printvalue(value:T){
        println(value)
    }
}
//调用类型方法
NewEnum<Int>.printvalue(5)           //输出整型
NewEnum<String>.printvalue("Hello")  //输出字符串
NewEnum<Float>.printvalue(10.2222)   //输出浮点型
```

此时运行程序, 会看到如下的结果:

```
5
Hello
10.2222003936768
```

2. 泛型结构

泛型结构定义的形式如下:

```
struct 结构名称<T>{
    ...
}
```

创建某一类型的结构对象的形式如下:

```
let/var 对象名=结构名称<数据类型>()
```

【示例 4-69】 以下将实现栈的进栈和出栈功能。代码如下:

```
//定义泛型结构 Stack
struct Stack<T> {
    var items = [T]()
    //定义可变方法
    mutating func push(item: T) {
        items.append(item)
    }
}
```



```

//定义可变方法
mutating func pop() -> T {
    return items.removeLast()
}
}
var stackOfStrings = Stack<String>()
println("入栈")
//进栈
stackOfStrings.push("one")
stackOfStrings.push("two")
stackOfStrings.push("three")
stackOfStrings.push("four")
for index in stackOfStrings.items {
    println(index)
}
println("出栈")
//出栈
let fromTheTop = stackOfStrings.pop()
for index in stackOfStrings.items {
    println(index)
}

```

此时运行程序，会看到如下的结果：

```

入栈
one
two
three
four
出栈
one
two
three

```

3. 泛型类

泛型类的定义形式如下：

```

class 类名<T>{
    ...
}

```

创建某一类型的类对象的形式如下：

```

let/var stringclass=类名<数据类型>()

```

【示例 4-70】 以下将定义一个泛型类，让它可以输出任意类型的内容。代码如下：

```

class NewClass<T>{
    //定义实例方法 printvalue
    func printvalue(value:T){
        println(value)
    }
}
let stringclass=NewClass<String>()           //实例化对象
stringclass.printvalue("Swift")              //输出内容
let intclass=NewClass<Int>()                  //实例化对象
intclass.printvalue(10)                       //输出内容

```

此时运行程序，会看到如下的结果：

```
Swift
10
```

4.8.3 具有多个类型参数的泛型

在泛型中可以拥有多个类型泛型，这些类型参数需要使用逗号分隔开。

【示例 4-71】 以下将在一个泛型中使用 3 个泛型类型。代码如下：

```
class NewClass<T,U,W>{
    func printvalue(value1:T,value2:U,value3:W) {
        println(value1)
        println(value2)
        println(value3)
    }
}
let newclass=NewClass<Int,String,Double>()
newclass.printvalue(100,value2:"Swift",value3:88.88)
```

此时运行程序，会看到如下的结果：

```
100
Swift
88.88
```

4.8.4 类型约束

类型约束指定了一个必须继承自指定类的类型参数，或者遵循一个特定的协议或协议构成。对于类型约束，开发者可以在类型参数名后面写一个类型约束，并通过冒号:将其分隔。对于泛型函数的类型约束的定义形式如下：

```
func 函数名称<T: SomeClass, U: SomeProtocol,...>(someT: T, someU: U) {
    ...
}
```

上面这个函数中有两个类型参数。第一个类型参数 T 必须是 SomeClass 子类的类型约束；第二个类型参数 U 必须遵循 SomeProtocol 协议的类型约束。对于泛型函数的类型约束同时也适用于其他泛型类型。

【示例 4-72】 以下将定义一个泛型函数 findIndex()，此泛型函数有一个类型约束，即 T 必须要遵守 Equatable 协议。此函数的功能是查找包含一定数据类型值的数组。代码如下：

```
//定义泛型函数，此函数的类型参数 T 必须要遵守 Equatable 协议
func findIndex<T: Equatable>(array: [T], valueToFind: T) -> Int? {
    //遍历数据
    for (index, value) in enumerate(array) {
        //判断是否相等
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```



```
//判断在数据[3.14159, 0.1, 0.25]中是否包含 9.3
let doubleIndex = findIndex([3.14159, 0.1, 0.25], 9.3)
println(doubleIndex)
//判断在数据["Mike", "Malcolm", "Andrea"]中是否包含"Andrea"
let stringIndex = findIndex(["Mike", "Malcolm", "Andrea"], "Andrea")
println(stringIndex)
```

此时运行程序，会看到如下的结果：

```
nil
Optional(2)
```

4.8.5 关联类型

当定义一个协议时，有时候声明一个或多个关联类型作为协议定义的一部分是非常有用的。一个关联类型给定作用于协议类型的一个节点（或别名）。本小节将讲解关联类型的定义、扩展已存在类型为关联类型等内容。

1. 定义关联类型

定义关联类型需要使用 `typealias` 关键字，其语法形式如下：

```
typealias 类型名
```

【示例 4-73】 以下使用关联类型实现进栈和出栈的功能。代码如下：

```
//定义协议
protocol Container {
    typealias ItemType //定义关联类型
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}
//定义泛型结构，并遵守协议 Container
struct Stack<T>: Container {
    var items = [T]()
    //定义可变方法 push(), 实现进栈的功能
    mutating func push(item: T) {
        items.append(item)
    }
    //定义可变方法 pop(), 实现出栈的功能
    mutating func pop() -> T {
        return items.removeLast()
    }
    //定义可变方法 append(), 实现进栈的功能
    mutating func append(item: T) {
        self.push(item)
    }
    //定义计算属性
    var count: Int {
        return items.count
    }
    //定义下标脚本
    subscript(i: Int) -> T {
        return items[i]
    }
}
```



```

}
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var pushstack = Stack<Int>()
        //进栈
        pushstack.push(1)
        pushstack.push(2)
        println("现在一共有 \(pushstack.count) 个元素")
        //出栈
        var popstack=pushstack.pop()
        println("出栈后剩余 \(pushstack.count) 个元素")
    }
}

```

此时运行程序，会看到如下的结果：

```

现在一共有 2 个元素
出栈后剩余 1 个元素

```

2. 扩展已存在类型为关联类型

扩展除了可以向已有的类型中添加方法和属性等外，还可以将已存在的类型扩展为关联类型。

【示例 4-74】 以下将首先定义一个协议 `Container`，在此协议中定义了关联类型 `ItemType`。然后，定义一个泛型结构 `Stack`，接着再扩展 `Stack` 结构，并遵守协议 `Container`。实现栈的一些功能。代码如下：

```

//定义协议
protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
}
//定义泛型结构 Stack
struct Stack<T>{
    var items = [T]()
}
//扩展泛型结构，并遵守 Container 协议
extension Stack: Container {
    //定义可变方法 push()，实现进栈的功能
    mutating func push(item: T) {
        items.append(item)
    }
    //定义可变方法 pop()，实现出栈的功能
    mutating func pop() -> T {
        return items.removeLast()
    }
    //定义可变方法 append()，实现进栈的功能
    mutating func append(item: T) {
        self.push(item)
    }
    //定义计算属性 count，实现栈中元素个数的计算
    var count: Int {
        return items.count
    }
}

```

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        var pushstack = Stack<Int>()
        //进栈
        pushstack.push(1)
        pushstack.push(2)
        pushstack.push(3)
        println("现在一共有 \(pushstack.count) 个元素")
        //出栈
        var popstack=pushstack.pop()
        println("出栈后剩余 \(pushstack.count) 个元素")
    }
}
```

此时运行程序，会看到如下的结果：

```
现在一共有 3 个元素
出栈后剩余 2 个元素
```

3. 约束关联类型

类型约束能够确保类型符合泛型函数或类的定义约束。对于关联类型来说，约束也是非常有用的。关联类型的约束需要使用 `where` 语句。一个 `where` 语句不但可以使一个关联类型遵循一个特定的协议进行运行，并且可以使特定类型的参数和关联类型是相同的。开发者可以写一个 `where` 语句，紧跟在类型参数列表后面。`where` 语句后跟一个或者多个针对关联类型的约束，以及一个或多个类型和关联类型间的等价关系。在泛型函数中约束关联类型的语法形式如下：

```
func 函数名<T where 约束内容 1, 约束内容 2, ...>
```

其中，`T` 表示类型参数名。对于泛型类型来说，它的约束关联类型和泛型函数的约束关联类型的语法形式一样。

【示例 4-75】 以下将使用约束关联类型检查两个 `Container` 实例是否包含相同顺序的元素。代码如下：

```
//定义协议 Container
protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}
//定义泛型结构 Stack
struct Stack<T>: Container {
    var items = [T]()
    //定义可变方法 push，实现进栈的功能
    mutating func push(item: T) {
        items.append(item)
    }
    //定义可变方法 pop，实现出栈的功能
    mutating func pop() -> T {
        return items.removeLast()
    }
    //定义可变方法 append，实现进栈的功能
```



```

mutating func append(item: T) {
    self.push(item)
}
//定义计算属性 count
var count: Int {
    return items.count
}
//定义下标脚本
subscript(i: Int) -> T {
    return items[i]
}
}
//约束关联类型
func allItemsMatch<C1: Container, C2: Container where C1.ItemType ==
C2.ItemType, C1.ItemType: Equatable>(someContainer: C1, anotherContainer:
C2) -> Bool {
    //检查两个 Container 的元素个数是否相同
    if someContainer.count != anotherContainer.count {
        return false
    }
    //检查两个 Container 相应位置的元素彼此是否相等
    for i in 0..

```

此时运行程序, 会看到如下的结果:

不是所有项都匹配

第 5 章 iPhone 游戏开发基础—— 记忆配对游戏

前面章节已经讲解了 Swift 的基础语法，从本章开始将以实战的形式，讲解如何开发游戏。记忆配对游戏是一类经典的益智游戏。它不仅可以锻炼玩家的眼力，还可以锻炼玩家的记忆力。本章将通过这个游戏讲解游戏主菜单、游戏配对和场景切换等技术。

5.1 游戏介绍

记忆配对游戏是经典的益智游戏，老少皆宜。游戏开始时，提供许多反扣的卡牌。玩家可以连续将卡牌翻转过来。当连续的两张牌一样时，就自动消除；否则，这两张牌会被重新反扣回去。这样的游戏，通常包括以下几个模块。

1. 主菜单模块

主菜单提供一个基本的界面，帮助玩家操作，如图 5.1 所示。轻拍 Play Game 按钮，可以进入配对游戏的界面；轻拍 List Scene 按钮，可以进入分数榜单的界面；轻拍 About Scene 按钮，可以进入关于游戏的界面。

2. 配对模块

配对模块是程序最重要的模块。它提供了游戏的界面，如图 5.2 所示。同时，它会负责对玩家的操作做出响应。



图 5.1 主菜单

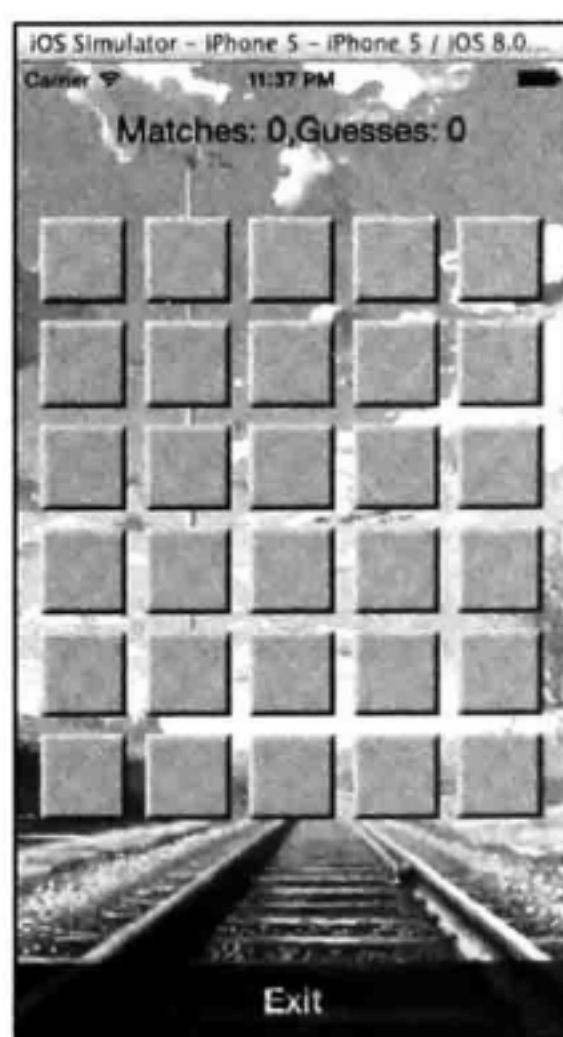


图 5.2 配对游戏界面

3. 分数榜单模块

分数榜单模块提供了显示玩家分数的界面，如图 5.3 所示。

4. 关于游戏的模块

关于游戏的模块提供了此游戏的玩法和说明等内容，如图 5.4 所示。

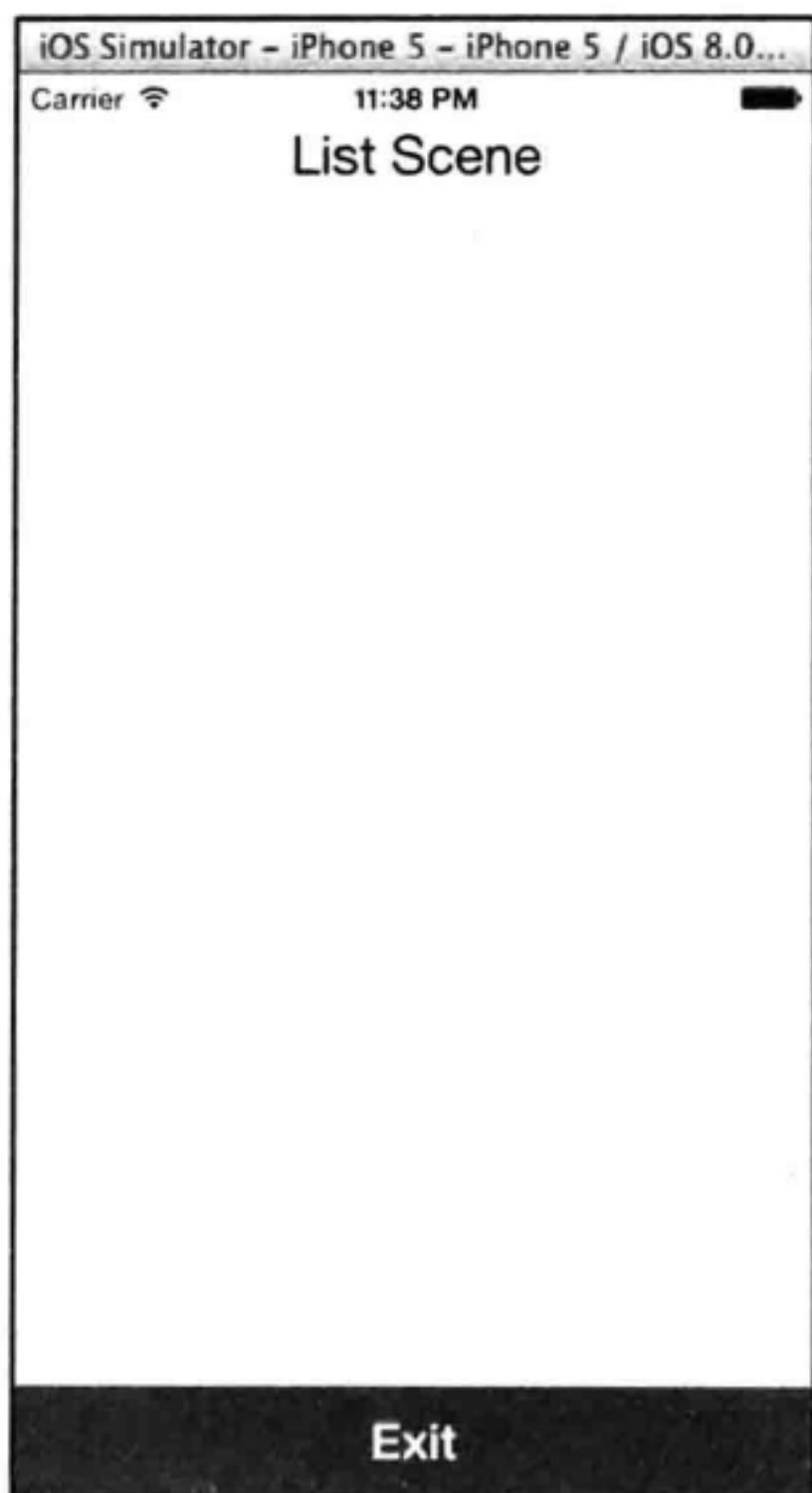


图 5.3 分数榜单



图 5.4 关于游戏

5.2 开发游戏之前的准备工作

本节将讲解一些在开发游戏之前的准备工作。

5.2.1 创建项目

和其他的 iOS 应用程序一样，在开发一款游戏之前，首先需要创建一个项目。在此项目中保存了开发游戏的代码和资源等内容。在本章中，需要创建一个 Single View Application 模板类型的项目，命名为 MatchingGame。它的具体步骤如下。

(1) 单击 Dock 中的 Xcode，弹出 Welcome to Xcode 对话框。

(2) 选择其中的 Create a new Xcode project 选项，弹出 Choose a template for your new project:对话框，如图 5.5 所示。

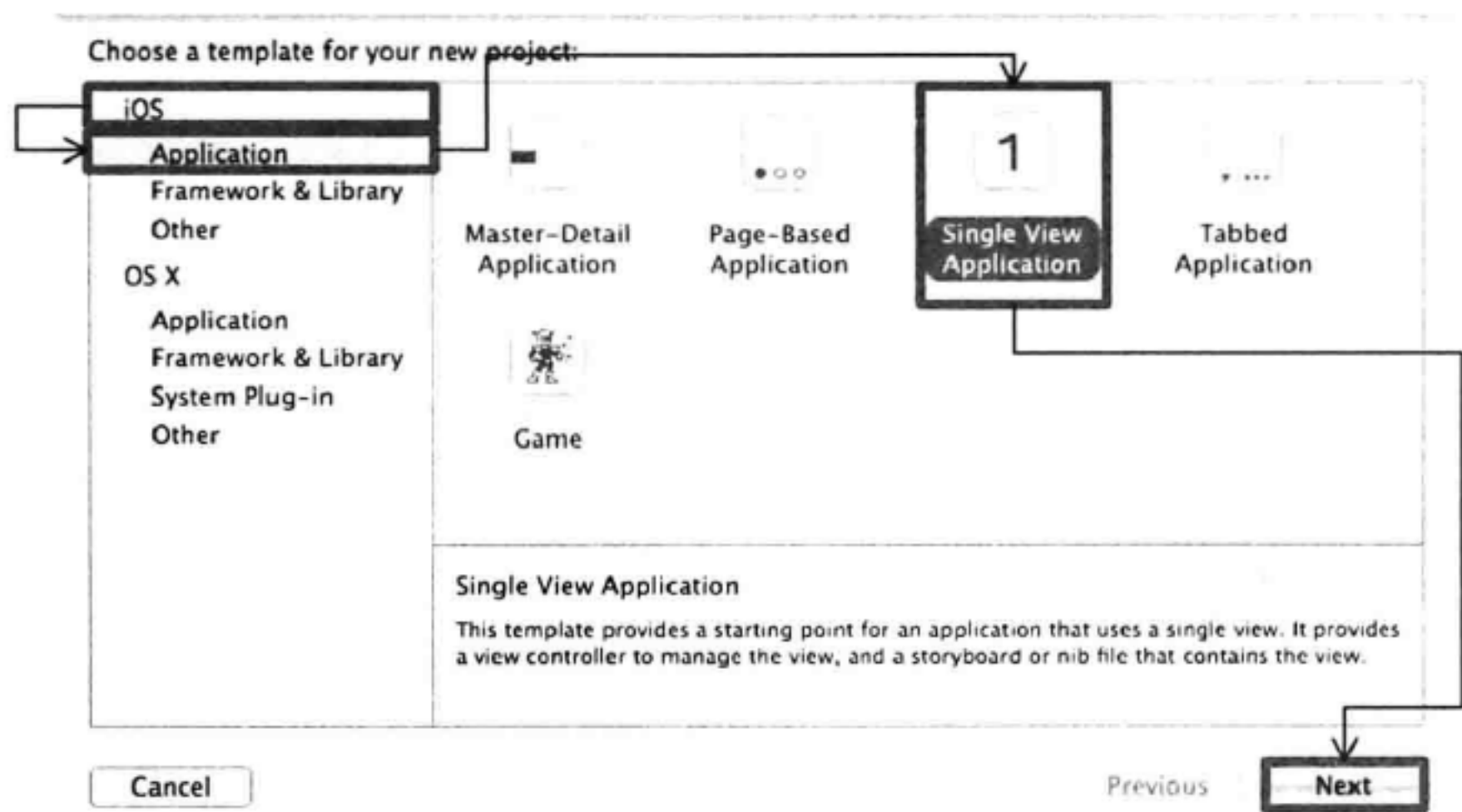


图 5.5 操作步骤 1

(3) 选择 iOS|Application 中的 Single View Controller 模板，然后单击 Next 按钮，弹出 Choose options for your new project:对话框，如图 5.6 所示。

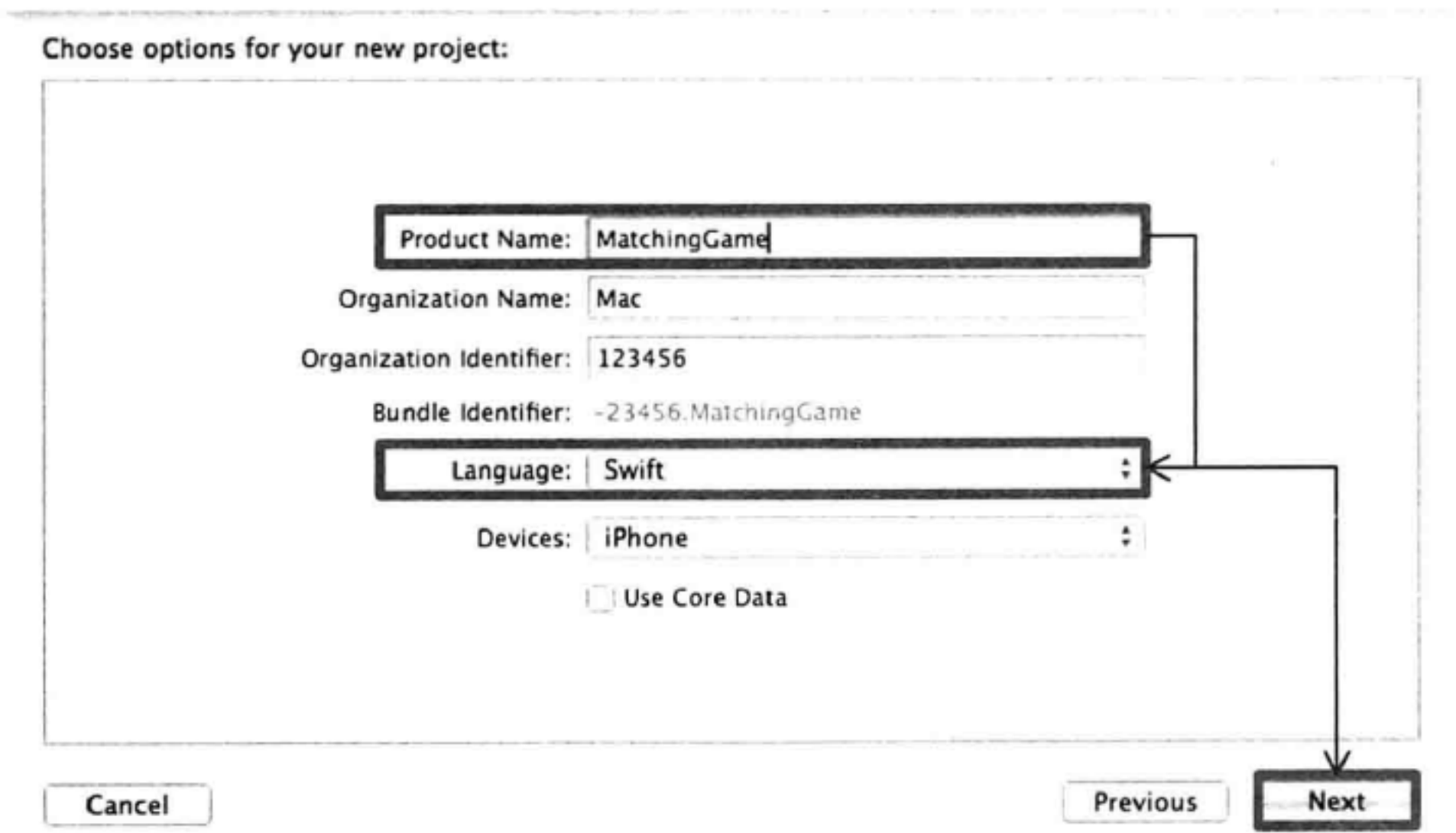


图 5.6 操作步骤 2

(4) 在此对话框中输入项目的名称 MatchingGame，将程序语言选择为 Swift。然后单击 Next 按钮，弹出选择项目保存位置的对话框。在此对话框中单击 Create 按钮后，一个名为 MatchingGame 的项目就创建好了。我们就可以在此项目中开发记忆配对游戏。

5.2.2 添加图像

为了让记忆配对游戏的界面看起来更加漂亮，需要在创建的项目，即 MatchingGame 中添加一些图像。这些图像有 back.png、blank.png、backdrop1.jpg、backdrop2.jpg、backdrop3.jpg、icons01.png~icons15.png。添加图像的具体操作步骤如下所述。

(1) 右击 Supporting Files 文件夹，弹出快捷菜单，如图 5.7 所示。

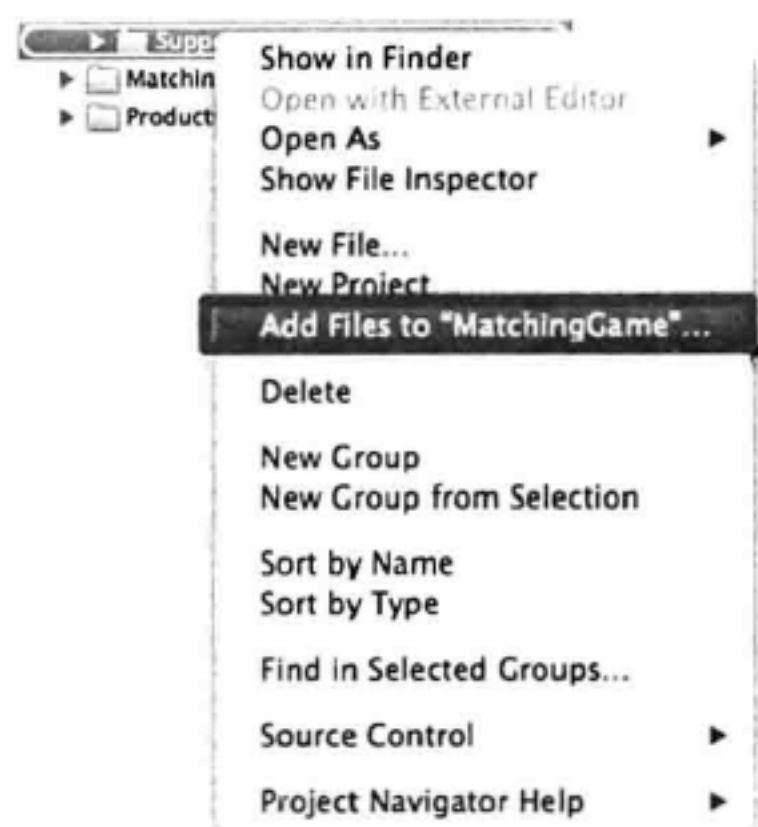


图 5.7 操作步骤 1

(2) 选择 Add Files to "MatchingGame"...命令，弹出选择文件对话框，如图 5.8 所示。



图 5.8 操作步骤 2

(3) 选择需要添加的图像后，单击 Add 按钮，实现图像的添加。添加后的图像就会显示在 Supporting Files 文件夹中。

5.3 主菜单模块

主菜单提供一个基本的界面。在一个主菜单中会拥有一组菜单项，它可以帮助玩家进

行操作。菜单项的实现方式有很多，如使用标签实现菜单项，使用按钮实现菜单项等。本节将使用按钮实现对主菜单中菜单项的设计。

双击将 Main.storyboard 文件打开，对主菜单的界面（界面也可以称之为主视图或者场景）进行设计，具体的操作步骤图像如下所述。

（1）选择 Show the File inspector 选项，将 Interface Builder Document 中的 Opens in 改为 Xcode 5.1，如图 5.9 所示。

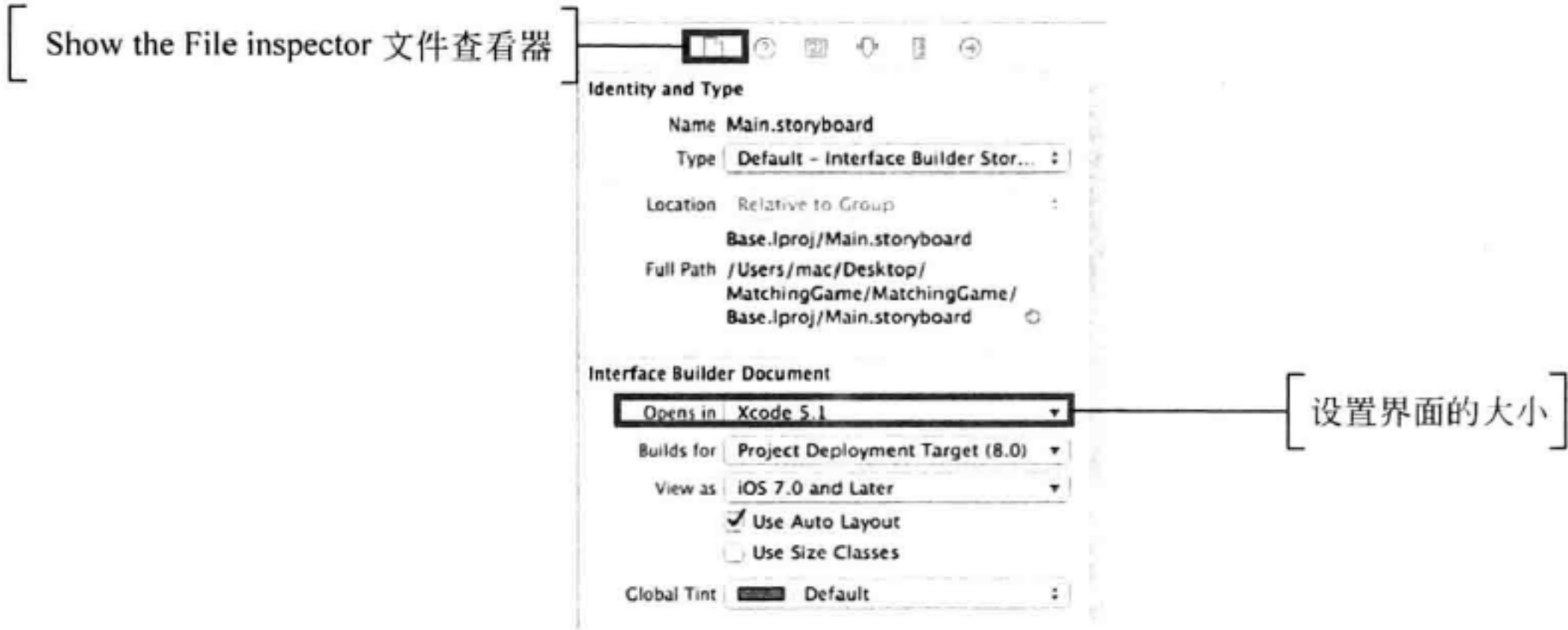


图 5.9 设置界面的大小

注意：由于 Xcode 6.0 的界面比较大，为了方便截图，将 Xcode 6.0 界面改为 Xcode 5.1 的界面。

Opens in 这一项开发者可以不去修改，在本书中，由于截图的缘故，所以将此项进行了修改（在后面的章节中实现的应用都将此项进行了修改），修改此项后，画板中的界面的尺寸会变为 Xcode 5.1 中的尺寸，如图 5.10 所示。

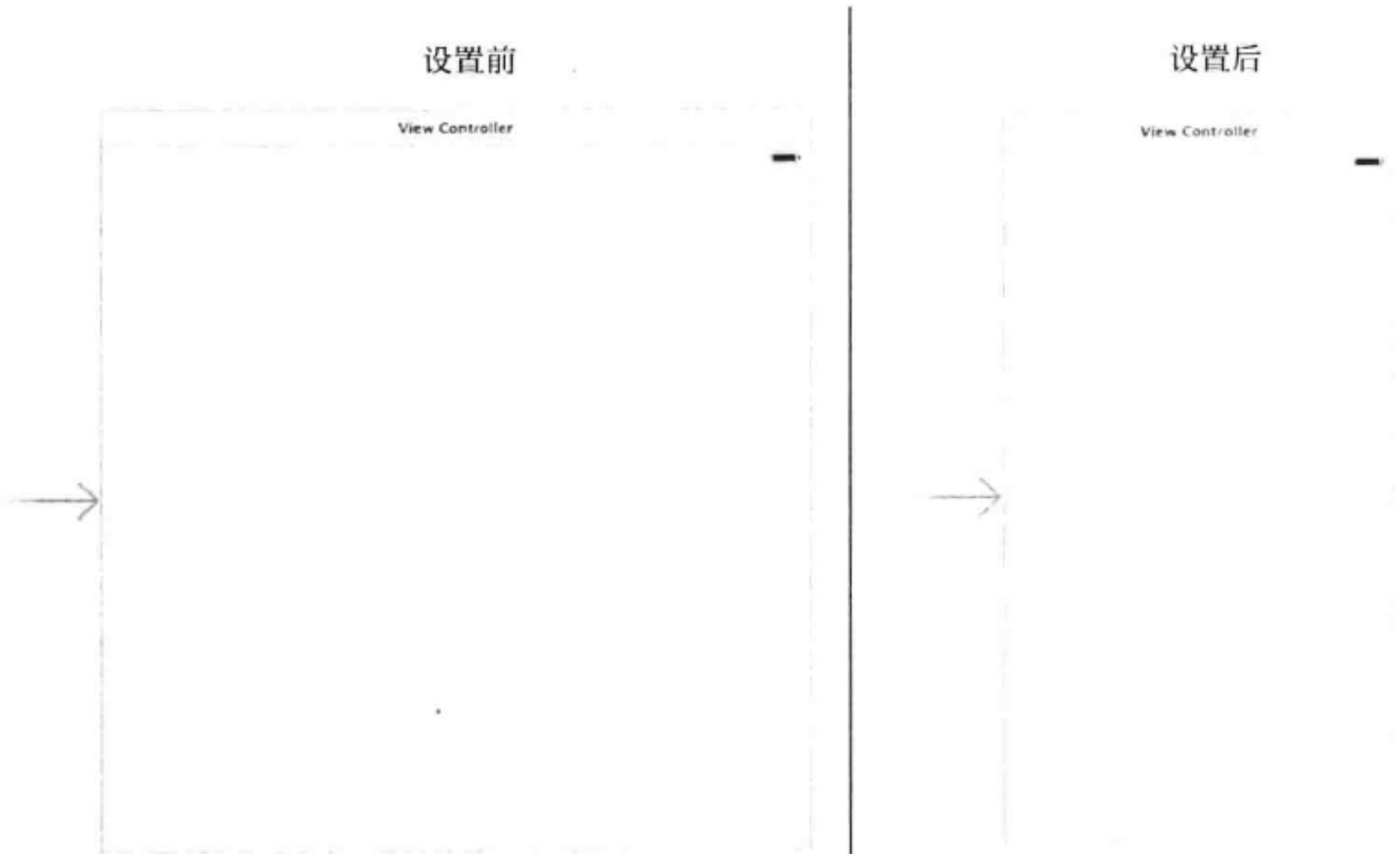


图 5.10 界面的大小设置

(2)对在画板中原本存在的 View Controller 视图控制器的界面进行设计,效果如图 5.11 所示。

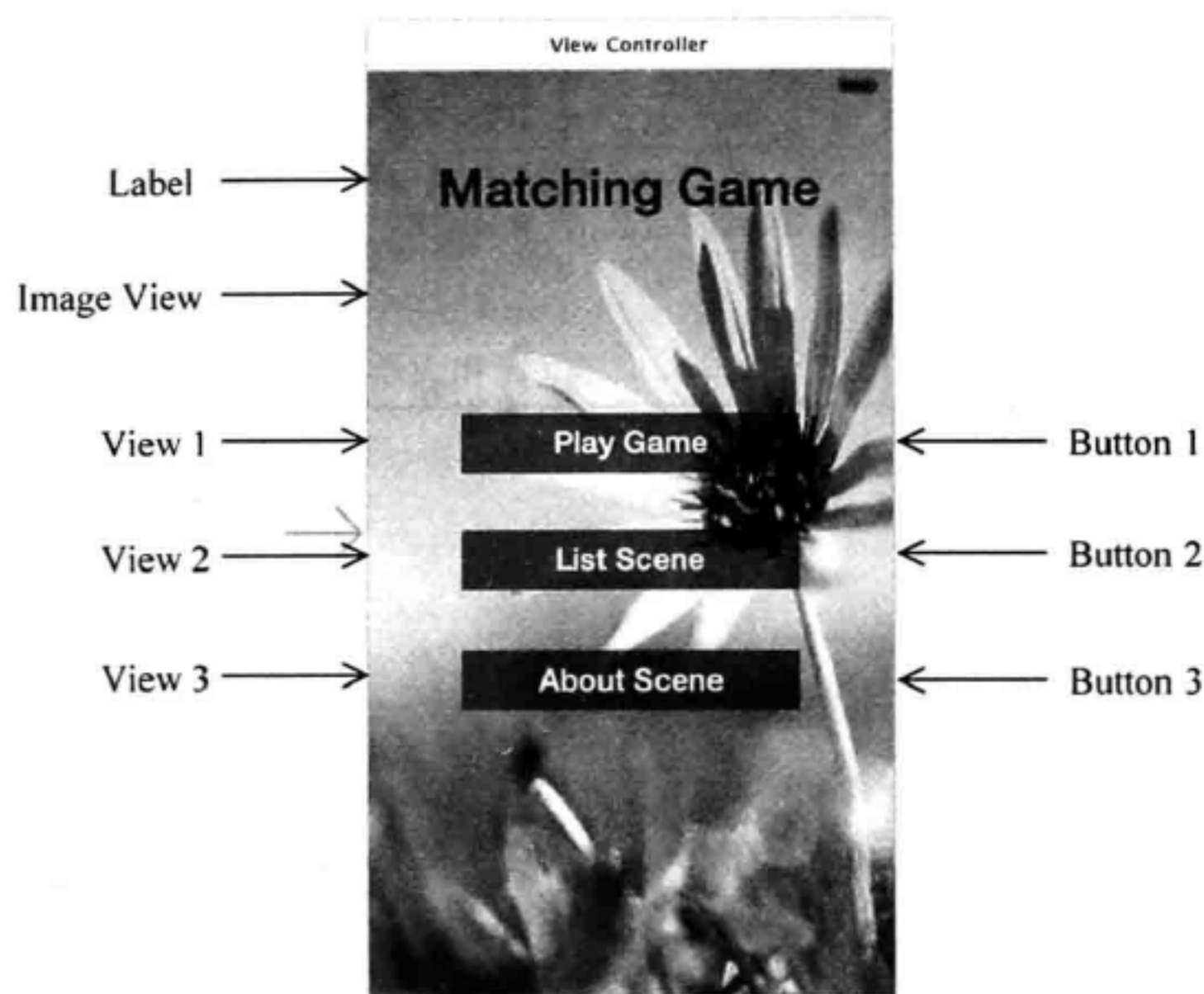


图 5.11 界面的效果

需要添加的视图对象，以及对它们的设置，如表 5-1 所示。

表 5-1 设置界面

视 图	设 置
Image View	Image: backdrop2.jpg 位置和大小: (0, 0, 320, 568)
Label	Text: Matching Game Font: System Bold 32.0 Alignment: 居中 位置和大小: (35, 52, 250, 42)
View1	Alpha: 0.5 Background: 黑色 位置和大小: (57, 211, 206, 37)
Button1	Title: Play Game Font: System Bold 19.0 Text Color: 白色 位置和大小: (40, 4, 127, 30)
View2	Alpha: 0.5 Background: 黑色 位置和大小: (57, 281, 206, 37)
Button2	Title: List Scene Font: System Bold 19.0 Text Color: 白色 位置和大小: (40, 4, 127, 30)

续表	
视 图	设 置
View3	Alpha: 0.5 Background: 黑色 位置和大小: (57, 351, 206, 37)
Button3	Title: About Scene Font: System Bold 19.0 Text Color: 白色 位置和大小: (40, 4, 127, 30)

在设菜单界面时需要注意以下两点。

1. 背景的生成

在图 5.11 中可以看到，在主菜单的界面上有一个背景，这个背景是采用了 Image View 图像视图对象实现的。首先，玩家需要在视图对象库中找到 Image View 对象，将其拖动到主菜单的界面中。然后，选择工具栏中的 Show the Attributes inspector 选项，即属性查看器，在其中找到 Image 属性，将此属性设置为 backdrop2.jpg，如图 5.12 所示。



图 5.12 设置 Image 属性

2. 菜单项的设置

按钮是玩家交互的最基础控件。对于它的使用首先需要从视图对象库中拖动 Button 按钮对象到主菜单的界面中，然后打开 Show the Attributes inspector 选项，即属性查看器对按钮的一些属性进行设置，如按钮的标题、标题颜色和字体大小等。

5.4 配对模块

配对模块是程序的最重要的模块。它提供了玩家游戏的界面，同时，它会负责对玩家的操作做出响应。本节将讲解有关配对模块的操作。

5.4.1 设计界面

在配对模块的游戏界面中，会拥有很多的卡牌，这些卡牌可以使用按钮进行实现。以下是游戏界面的设计步骤。

在视图对象库中拖动 View Controller 视图控制器对象到画布中，然后对此视图控制器的界面进行设计，效果如图 5.13 所示。

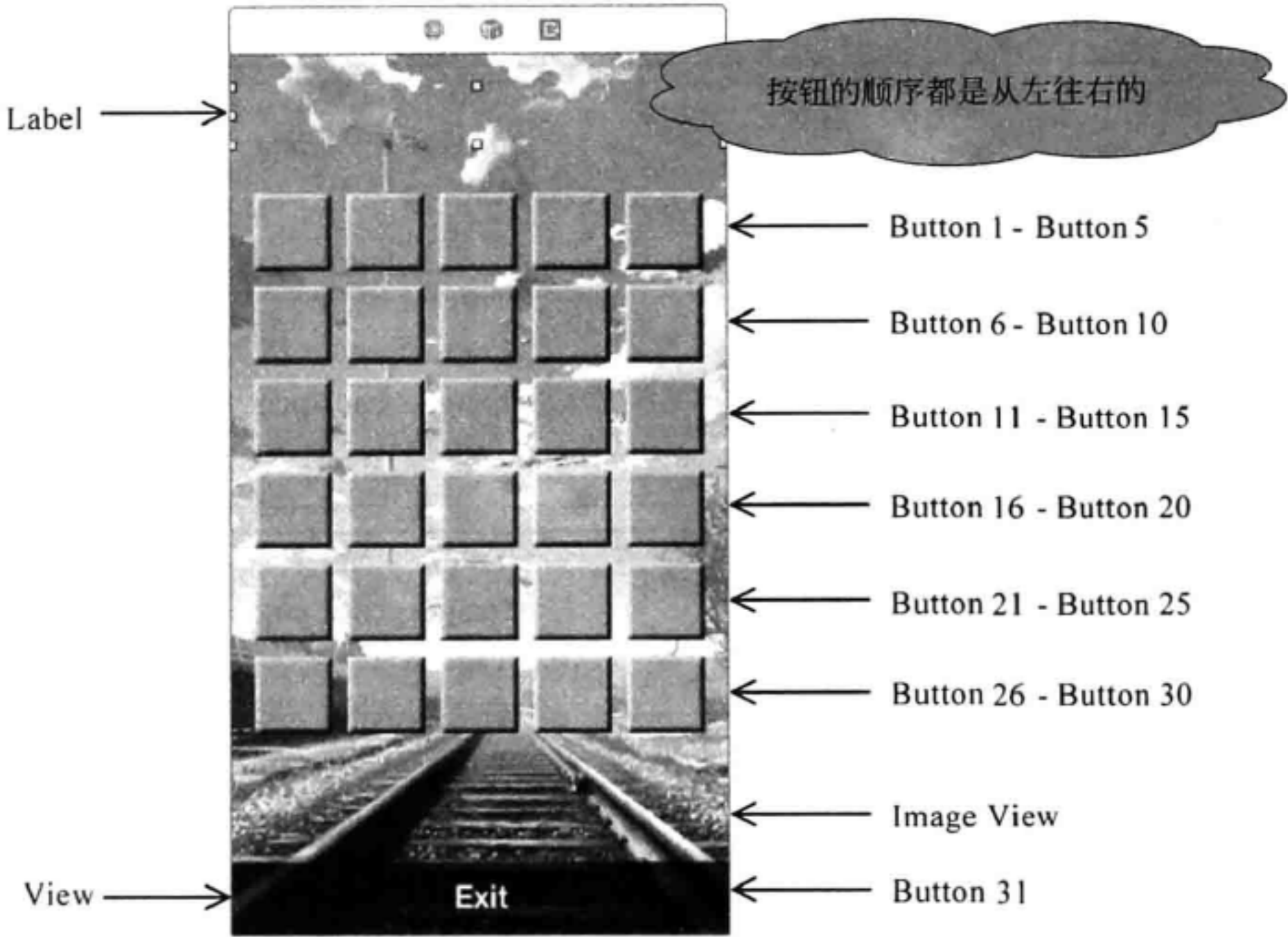


图 5.13 界面的效果

需要添加的视图对象，以及对它们的设置，如表 5-2 所示。

表 5-2 设置界面	
视 图	设 置
Image View	Image: backdrop1.jpg 位置和大小: (0, 0, 320, 568)
Label	Text: (空) Font: System 20.0 Alignment: 居中 位置和大小: (0, 20, 320, 39)
Button1	Title: (空) Image: back.png 位置和大小: (15, 90, 50, 50)
Button2	Title: (空) Image: back.png Tag: 1 位置和大小: (75, 90, 50, 50)

续表

视 图	设 置
Button3	Title: (空) Image: back.png Tag: 2 位置和大小: (135, 90, 50, 50)
Button4	Title: (空) Image: back.png Tag: 3 位置和大小: (195, 90, 50, 50)
Button5	Title: (空) Image: back.png Tag: 4 位置和大小: (255, 90, 50, 50)
Button6	Title: (空) Font: System 18.0 Image: back.png Tag: 5 位置和大小: (15, 150, 50, 50)
Button7	Title: (空) Image: back.png Tag: 6 位置和大小: (75, 150, 50, 50)
Button8	Title: (空) Image: back.png Tag: 7 位置和大小: (135, 150, 50, 50)
Button9	Title: (空) Image: back.png Tag: 8 位置和大小: (195, 150, 50, 50)
Button10	Title: (空) Image: back.png Tag: 9 位置和大小: (255, 150, 50, 50)
Button11	Title: (空) Image: back.png Tag: 10 位置和大小: (15, 210, 50, 50)
Button12	Title: (空) Image: back.png Tag: 11 位置和大小: (75, 210, 50, 50)
Button13	Title: (空) Image: back.png Tag: 12 位置和大小: (135, 210, 50, 50)

续表

视 图	设 置
Button14	Title: (空) Image: back.png Tag: 13 位置和大小: (195, 210, 50, 50)
Button15	Title: (空) Image: back.png Tag: 14 位置和大小: (255, 210, 50, 50)
Button16	Title: (空) Image: back.png Tag: 15 位置和大小: (15, 270, 50, 50)
Button17	Title: (空) Image: back.png Tag: 16 位置和大小: (75, 270, 50, 50)
Button18	Title: (空) Image: back.png Tag: 17 位置和大小: (135, 270, 50, 50)
Button19	Title: (空) Image: back.png Tag: 18 位置和大小: (195, 270, 50, 50)
Button20	Title: (空) Image: back.png Tag: 19 位置和大小: (255, 270, 50, 50)
Button21	Title: (空) Image: back.png Tag: 20 位置和大小: (15, 330, 50, 50)
Button22	Title: (空) Image: back.png Tag: 21 位置和大小: (75, 330, 50, 50)
Button23	Title: (空) Image: back.png Tag: 22 位置和大小: (135, 330, 50, 50)
Button24	Title: (空) Image: back.png Tag: 23 位置和大小: (195, 330, 50, 50)

续表	
视 图	设 置
Button25	Title: (空) Image: back.png Tag: 24 位置和大小: (255, 330, 50, 50)
Button26	Title: (空) Image: back.png Tag: 25 位置和大小: (15, 390, 50, 50)
Button27	Title: (空) Image: back.png Tag: 26 位置和大小: (75, 390, 50, 50)
Button28	Title: (空) Image: back.png Tag: 27 位置和大小: (135, 390, 50, 50)
Button29	Title: (空) Image: back.png Tag: 28 位置和大小: (195, 390, 50, 50)
Button30	Title: (空) Image: back.png Tag: 29 位置和大小: (255, 390, 50, 50)
View	Alpha: 0.65 Background: 黑色 位置和大小: (0, 523, 320, 45)
Button31	Title: Exit Font: System Bold 20.0 Text: 白色 位置和大小: (78, 7, 165, 30)

注意：在此界面中按钮 1 ~ 按钮 30 用来实现记忆配对游戏中的卡牌功能。

5.4.2 卡牌的翻转

本小节中将实现在配对游戏中轻拍卡牌（这里的卡牌就是按钮）后实现翻转的功能。

1. 创建文件

对于轻拍实现翻转动画这一功能，需要使用代码实现。为了使代码便于管理，在编写游戏代码之前，首先需要新建一个 Swift File 模板类型的文件，并命名为 Game。创建 Game 文件的具体步骤如下所述。

(1) 选择菜单栏上的 File|New|File 命令，弹出 Choose a template for your new file:对话框，如图 5.14 所示。

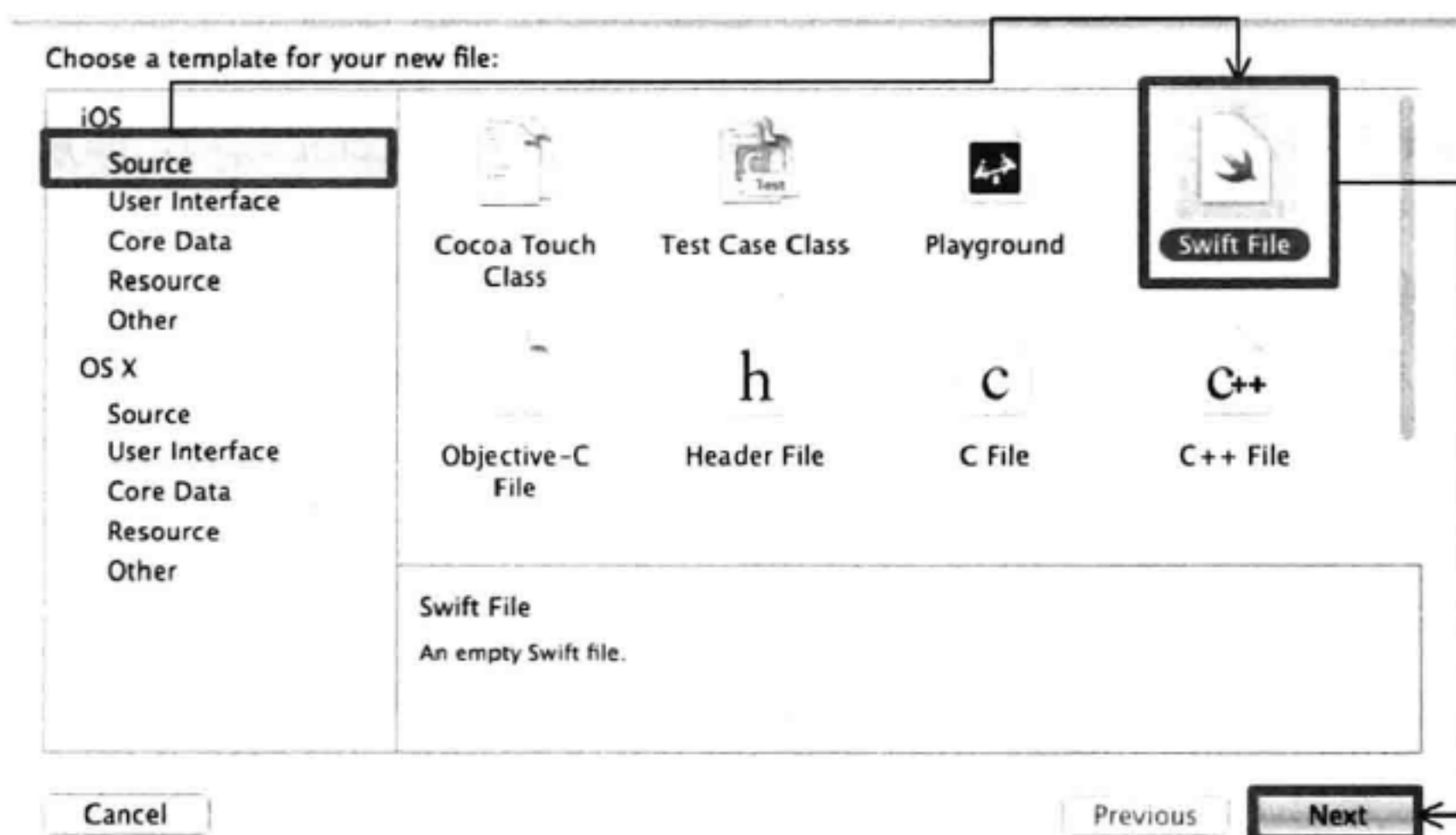


图 5.14 新建 swift 文件 1

(2) 选择 iOS|Source|Swift file 模板，然后单击 Next 按钮，弹出设置文件信息的对话框，其中包括文件名称和位置，如图 5.15 所示。

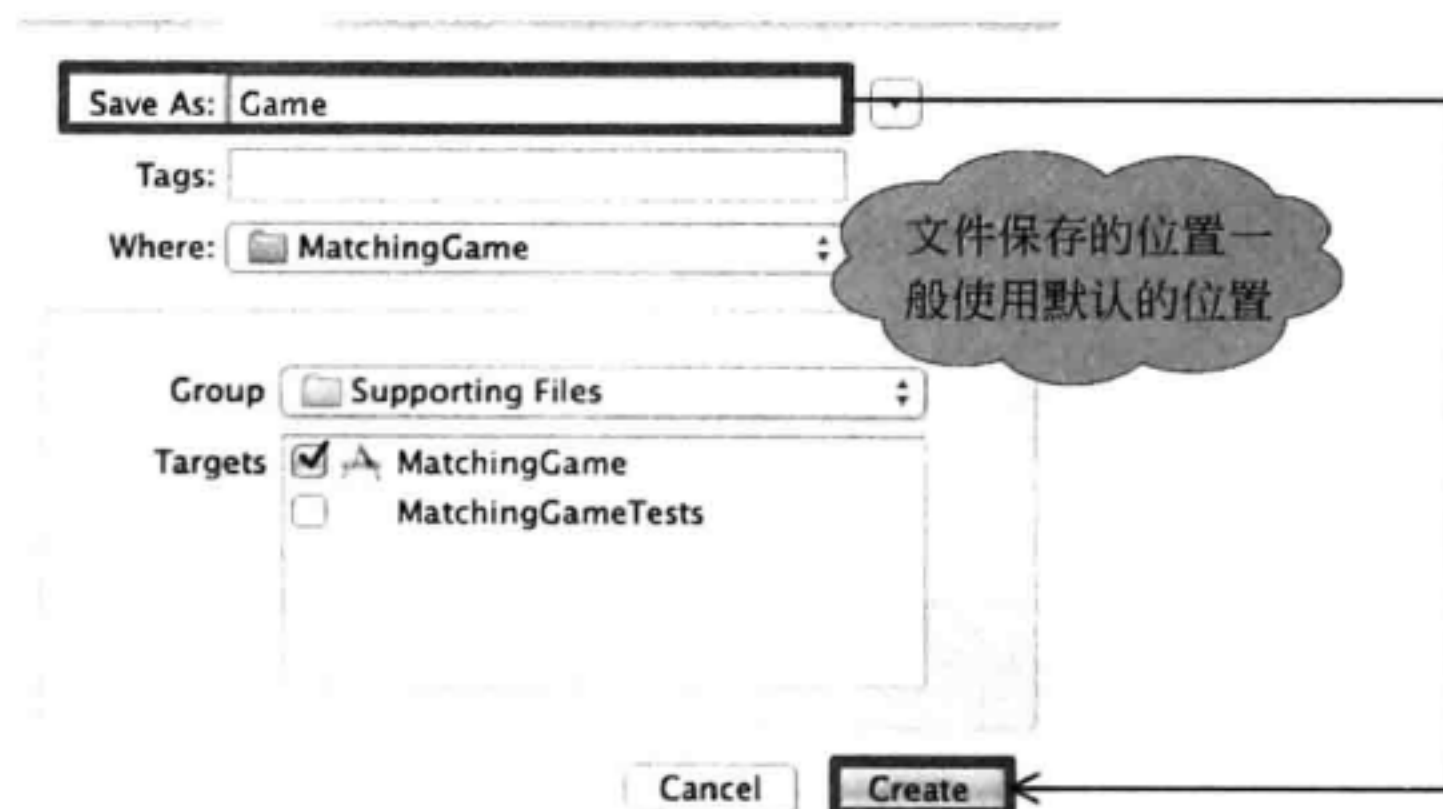


图 5.15 新建 swift 文件 2

(3) 输入文件名称后，单击 Create 按钮，此时就在创建的项目中新建了一个 Game.swift 的文件。

2. 创建空类

在添加文件后，需要创建一个基于 UIViewController 类的子类，代码如下：

```
import UIKit

class GameSceneViewController: UIViewController {

}
```


该类用于编写配对游戏的代码。

3. 视图控制器和类的关联

单击具有 31 个按钮的视图控制器，在此视图控制器中，选择界面上方的 Dock 中的 View Controller 图标。在工具窗口中的 Show the Identity inspector 选项，即属性查看器，将 Custom Class 下的 Class 设置为创建的 GameSceneViewController 类。这时在画布中的这个视图控制器就变为了 Game Scene View Controller 视图控制器，如图 5.16 所示。Dock 的位置如图 5.17 所示。

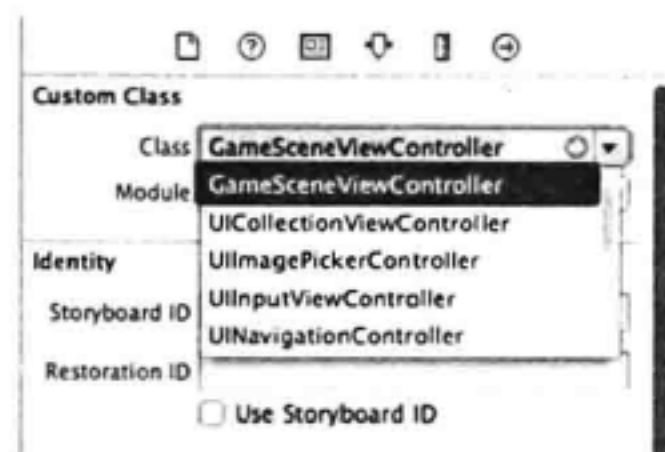


图 5.16 关联视图控制器

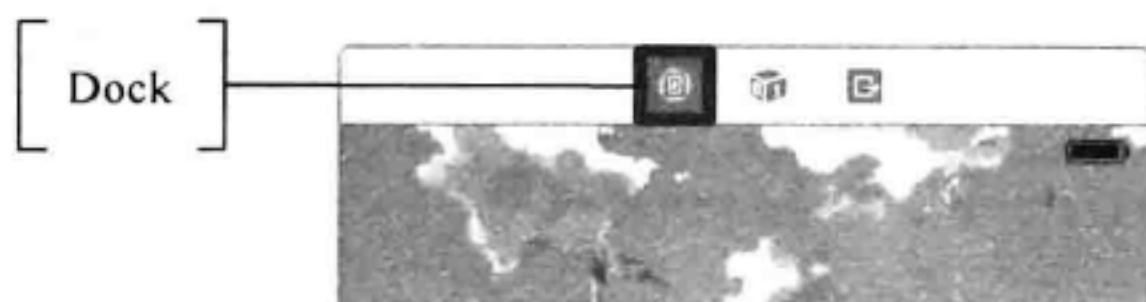


图 5.17 Dock

4. 轻拍卡牌后触发动作

当玩家轻拍卡牌（按钮）后，就会产生卡牌翻转的效果，而这种效果就是动作被触发所产生的（动作是一种方法）。在 iPhone 游戏开发中，对于动作的触发，可以使用关联来实现。以下是动作关联的具体步骤：

（1）使用调整窗口中的工具，将 Xcode 的界面调整为如图 5.18 所示的效果。这一过程在前面的章节中讲解过。

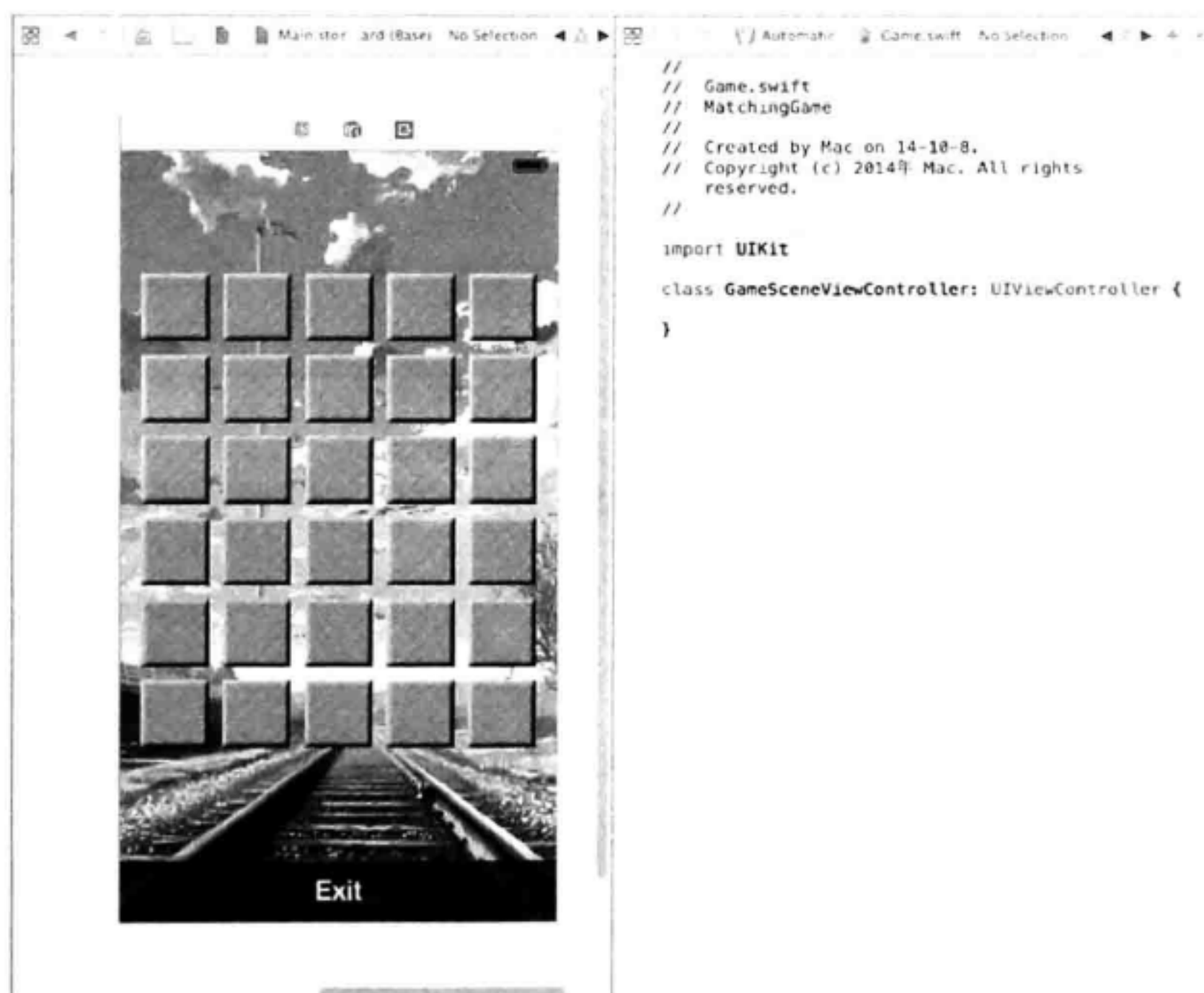


图 5.18 操作步骤 1

(2) 按住 Ctrl 键拖动界面中的按钮对象，这时会出现一个蓝色的线条，将这个蓝色的线条拖动到 Game.swift 文件中，如图 5.19 所示。

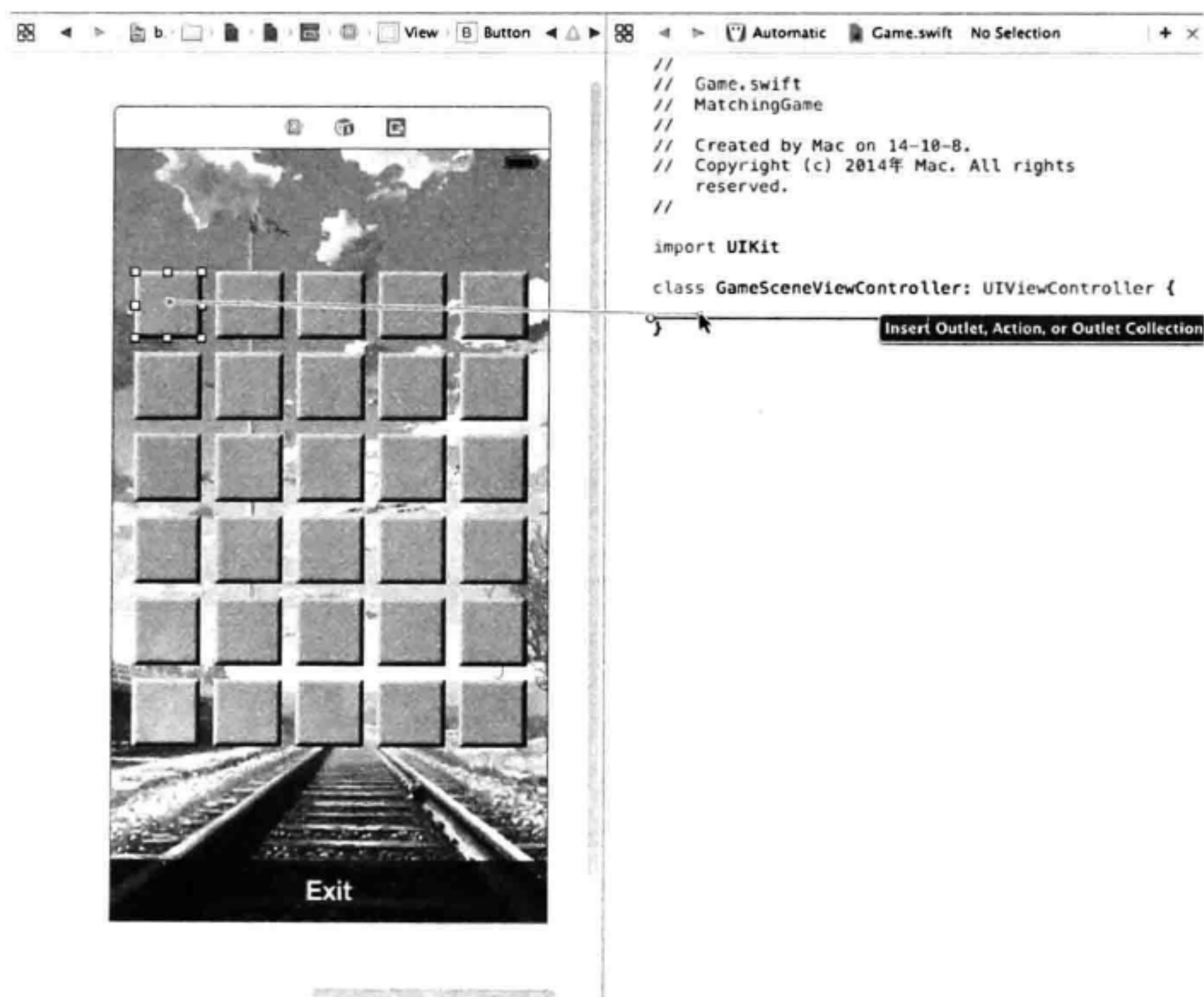


图 5.19 操作步骤 2

(3) 松开鼠标后，会弹出一个声明和关联插座变量在一起进行的对话框（在前面章节中讲解过），如图 5.20 所示。

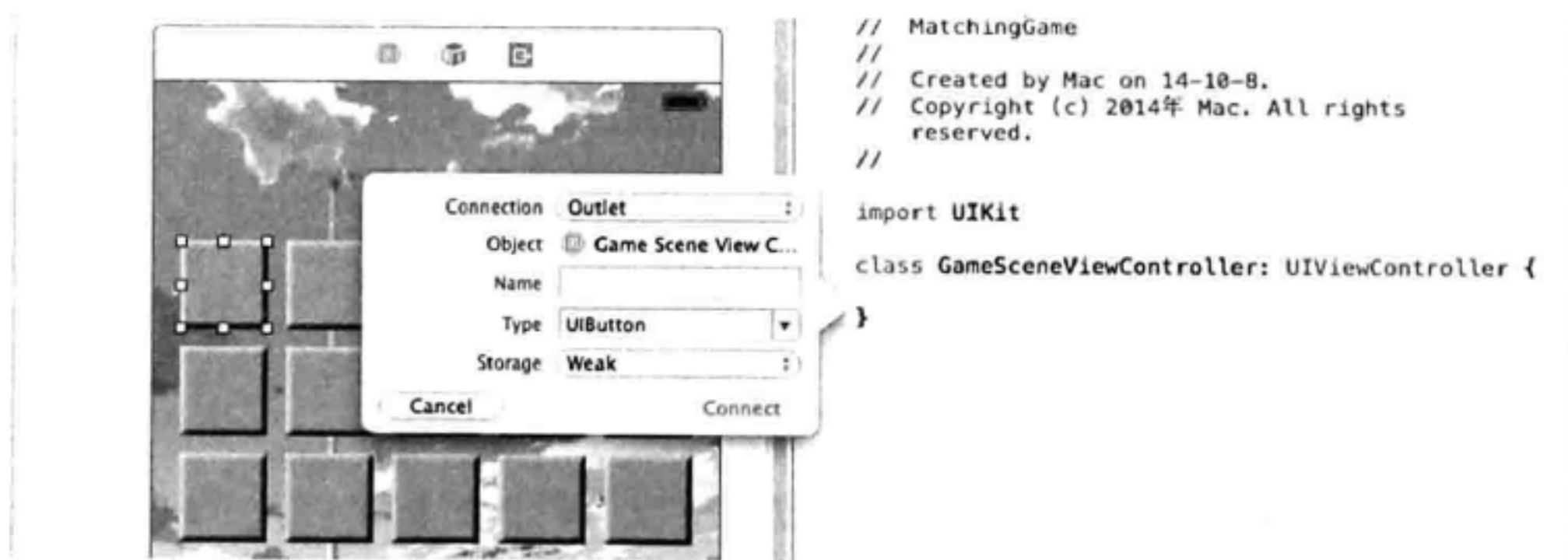


图 5.20 操作步骤 3

(4) 将 Connection 选项设置为 Action，将 Name 设置为 tileClicked，如图 5.21 所示。

 **注意：**这里的 Name 可以是任意的。

(5) 单击 Connect 按钮，会在 Game.swift 文件中看到如图 5.22 所示的代码。

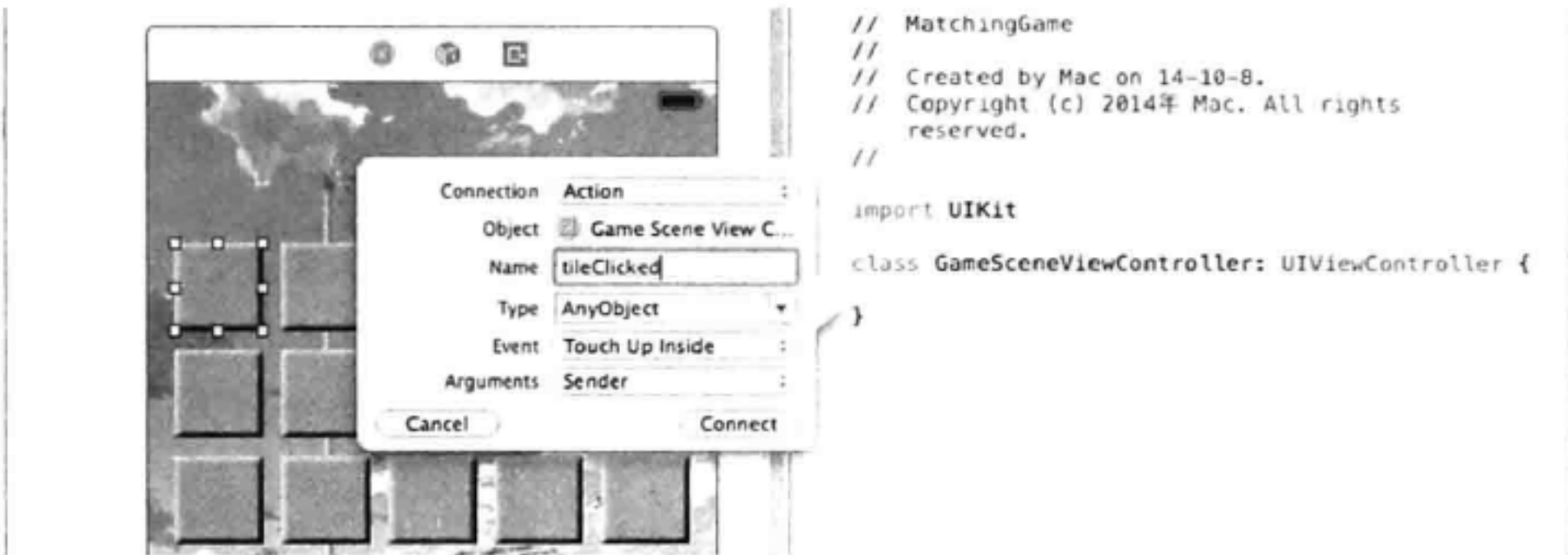


图 5.21 操作步骤 4

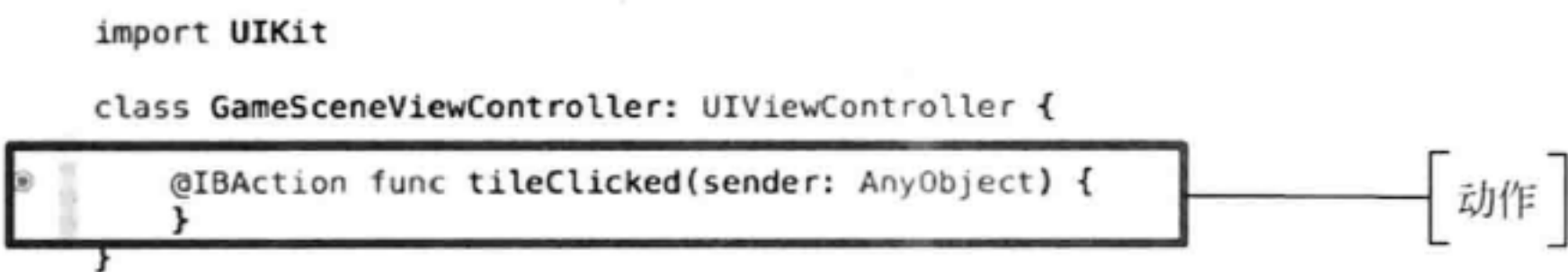


图 5.22 动作

此时，当玩家轻拍卡牌后，一个叫 tileClicked()的方法就会被触发。

注意：以上这一种方式是动作声明和关联一起进行的，还有一种先声明动作后关联的方式。声明动作可以使用关键字 IBAction。该关键字可以告诉故事面板的界面，此方法是一个操作，且可以被某个控件触发。声明动作的语法形式如下：

```
@IBAction func 动作名(参数:参数类型) {  
}
```

如图 5.23 所示，就是在 Game.swift 文件中编写的动作的声明代码。

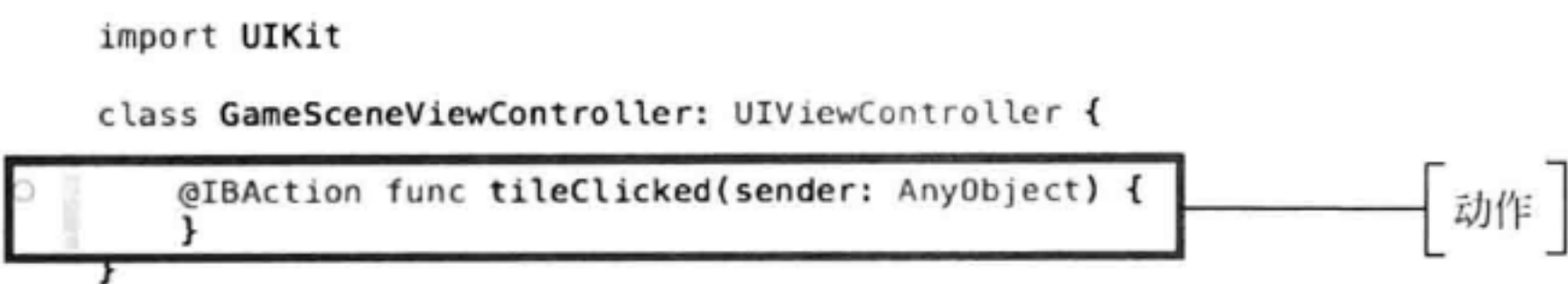


图 5.23 声明的动作

注意：在声明动作后，会在代码的前面出现一个空心的小圆圈，它表示此动作还未进行关联。

声明好动作后，就可以进行关联了，具体操作步骤如下所述。

- (1) 使用调整窗口中的工具，将 Xcode 的界面进行调整。将其调整为和图 5.18 一样的效果。
- (2) 按住 Ctrl 键拖动界面中的按钮对象，这时会出现一个蓝色的线条，将这个蓝色的线条和文件 Game.Swift 中的动作进行关联，如图 5.24 所示。

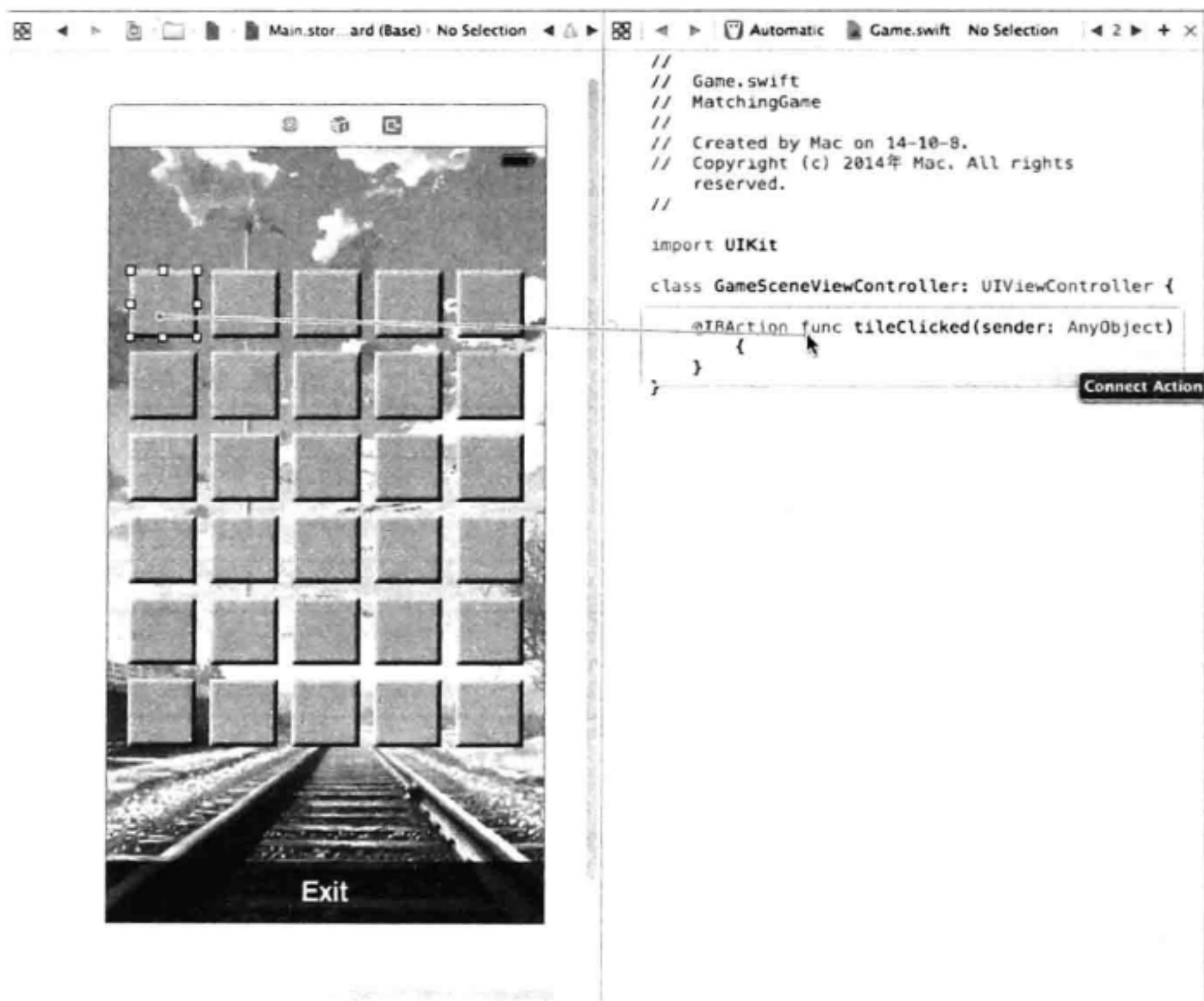


图 5.24 操作步骤

(3) 松开鼠标后，按钮对象就与动作成功的关联在一起了，此时动作前面的空心小圆圈就变为了实心的小圆圈，它表示此动作已被关联，效果和图 5.22 一样。以上是将第一个按钮和动作进行了关联，在本章将需要将第 1 个按钮一直到第 30 个按钮都与此动作进行关联。关联好的动作可以在按钮对象的连接查看器中看到，如图 5.25 所示。



图 5.25 关联的动作

5. 实现卡牌的翻转

卡牌的翻转其实很简单，就是在轻拍按钮后改变卡通的显示图像。此功能的实现，需

要使用到 setImage()方法，其语法形式如下：

```
按钮对象.setImage(image: UIImage?, forState: UIControlState)
```

其中，Image 用来指定显示的图像，forState 用来指定控件的状态。这些状态如表 5-3 所示。

表 5-3 控件的状态

按钮的状态	解 释 说 明
Normal	常规状态
Highlighted	高亮状态
Disabled	禁用状态，不接受任何事件
Selected	选中状态
Application	应用程序标志
Reserved	为内部框架预留，可以不管它

以下是卡牌翻转的实现。

首先需要在创建的 GameSceneViewController 类中初始化一个 tiles 数组对象。在这个数组对象中保存了一系列的图像。然后在 tileClicked()方法中添加代码，实现对卡牌的翻转效果，添加的代码如下：

```
import UIKit
class GameSceneViewController: UIViewController {
    //初始化数组
    var tiles:NSMutableArray=NSMutableArray(objects:
        UIImage(named: "icons01.png"),
        UIImage(named: "icons01.png"),
        UIImage(named: "icons02.png"),
        UIImage(named: "icons02.png"),
        UIImage(named: "icons03.png"),
        UIImage(named: "icons03.png"),
        UIImage(named: "icons04.png"),
        UIImage(named: "icons04.png"),
        UIImage(named: "icons05.png"),
        UIImage(named: "icons05.png"),
        UIImage(named: "icons06.png"),
        UIImage(named: "icons06.png"),
        UIImage(named: "icons07.png"),
        UIImage(named: "icons07.png"),
        UIImage(named: "icons08.png"),
        UIImage(named: "icons08.png"),
        UIImage(named: "icons09.png"),
        UIImage(named: "icons09.png"),
        UIImage(named: "icons10.png"),
        UIImage(named: "icons10.png"),
        UIImage(named: "icons11.png"),
        UIImage(named: "icons11.png"),
        UIImage(named: "icons12.png"),
        UIImage(named: "icons12.png"),
        UIImage(named: "icons13.png"),
        UIImage(named: "icons13.png"),
        UIImage(named: "icons14.png"),
        UIImage(named: "icons14.png"),
        UIImage(named: "icons15.png"),
        UIImage(named: "icons15.png"))
}
```



```

// 点击按钮实现翻转
@IBAction func tileClicked(sender: AnyObject) {
    var senderID:Int=sender.tag; // 获取轻拍按钮的 tag 值
    var tileImage:UIImage=self.tiles.objectAtIndex(senderID) as UIImage // 设置图像
    sender.setImage(tileImage, forState: UIControlState.Normal) // 设置按钮的图像
}
}

```

如果开发者想要查看配对模块中此时的效果，该怎么办呢？此时运行程序之后会看到主菜单，不会看到进行配对的游戏场景。它的解决办法如下所述。

(1) 回到 Main.storyboard 文件中。

(2) 单击具有 31 个按钮的视图控制器，在此视图控制器中，选择界面上方的 Dock 中的 View Controller 图标，在工具窗口中的 Show the Attributes inspector 选项，即属性查看器选中，找到 Is Initial View Controller 复选框，将其选中，如图 5.26 所示。它的功能就是将此视图控制器变为运行应用程序后，除启动界面后出现的第一个界面，即初始视图控制器。

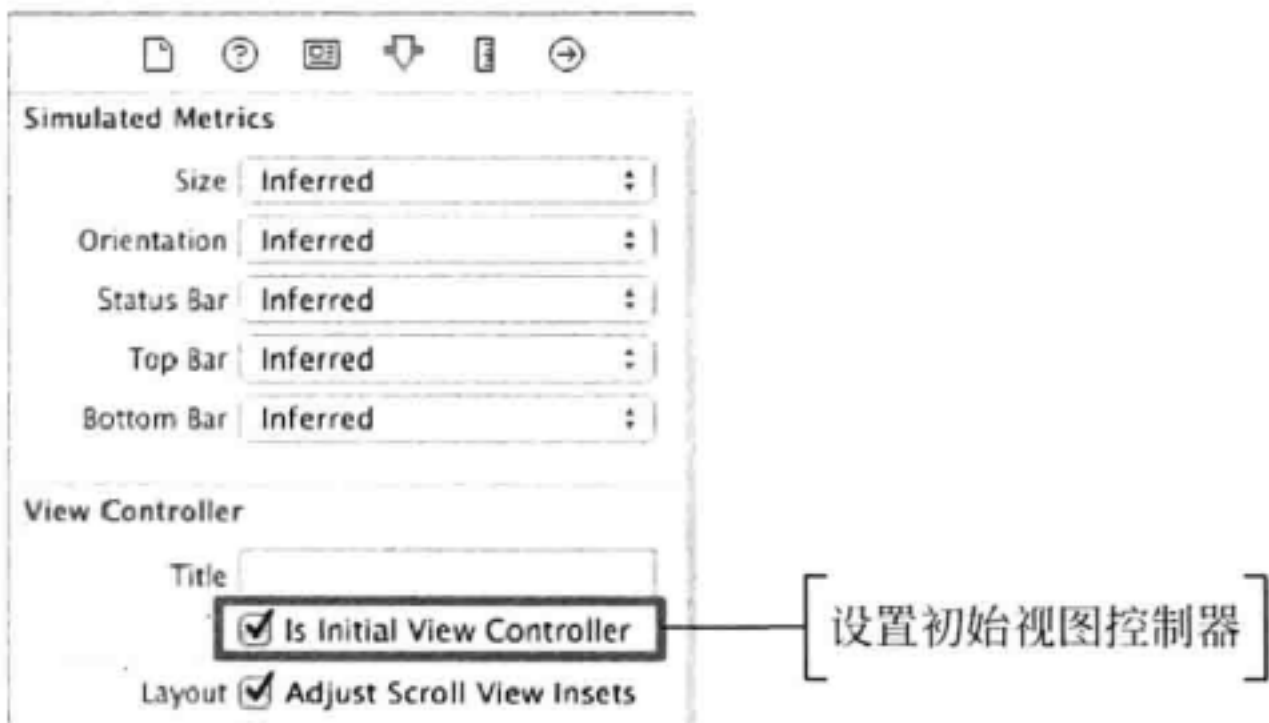


图 5.26 设置初始视图控制器

此时运行程序，可以看到如图 5.27 所示的效果。

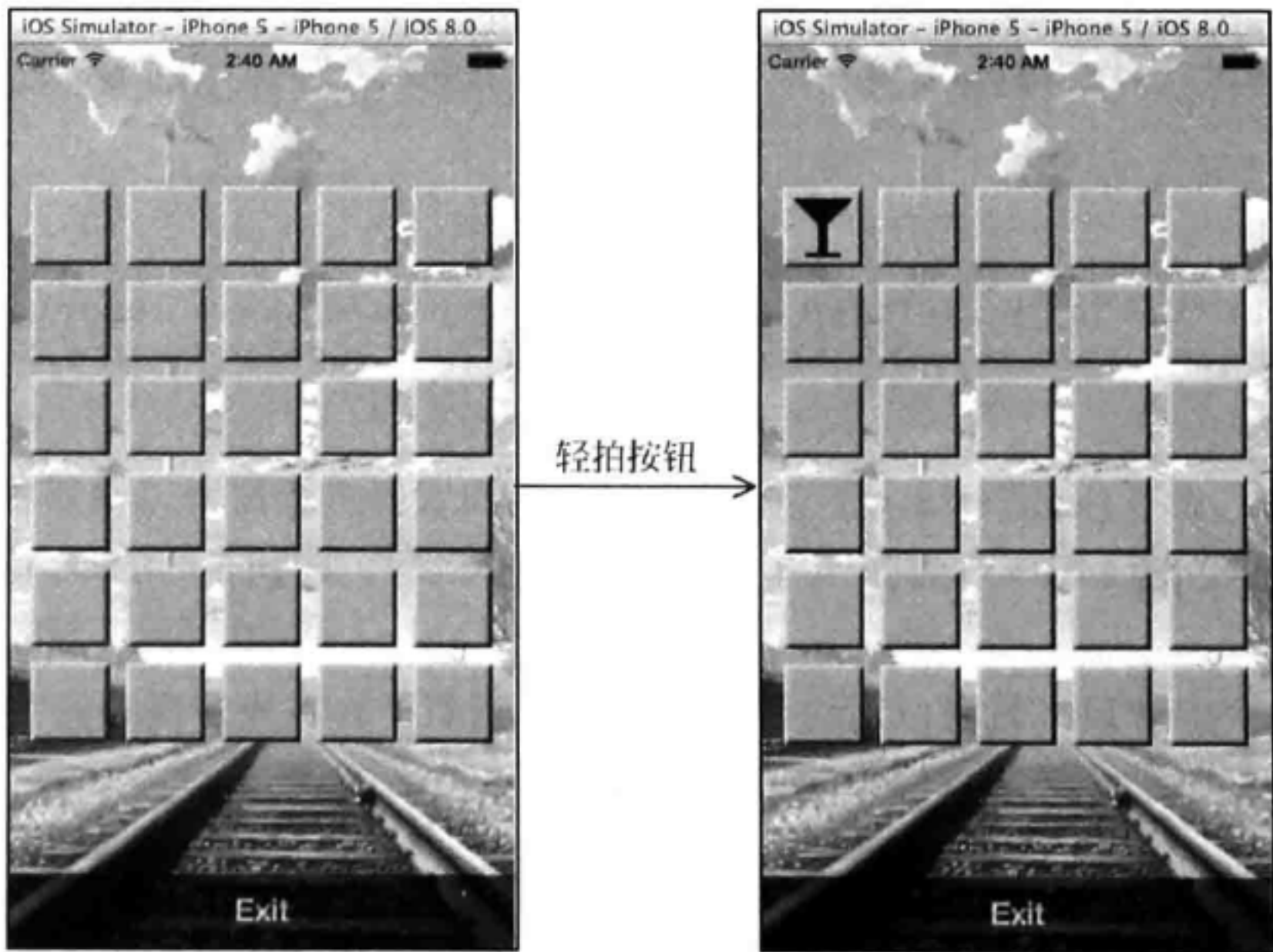


图 5.27 运行效果

5.5 核心功能——卡牌的配对

以下是卡牌配对的具体实现步骤。

5.5.1 翻转两个卡牌

在上一小节的翻转卡牌时，可以看到所有的卡牌都是可以进行翻转的。而在一个正规的配对游戏中，连续翻转的卡牌只能为两个，在翻转第 3 个卡牌时，是不应该被触发的。以下就是对这一个功能的实现。首先需要声明和定义变量，代码如下：

```
var isDisabled=false
var tileFlipped:NSInteger=0
var tile1:UIButton=UIButton()
var tile2:UIButton=UIButton()
```

然后，在 `tileClicked()` 动作中编写代码，实现翻转两个卡牌的功能，代码如下：

```
@IBAction func tileClicked(sender: AnyObject) {
    //判断轻拍的按钮是否禁用
    if(isDisabled==true){
        return
    }
    var senderID:Int=sender.tag;
    //判断翻转按钮的 tag 值是否大于 0，并且和当前轻拍按钮的 tag 值是否相等
    if(self.tileFlipped>=0&&senderID != self.tileFlipped){
        //翻转第二个卡牌
        self.tile2=sender as UIButton
        var lastImage:UIImage=self.tiles.objectAtIndex(self.tileFlipped)
        as UIImage
        var tileImage:UIImage=self.tiles.objectAtIndex(senderID) as UIImage
        sender.setImage(tileImage, forState: UIControlState.Normal)
        isDisabled=true //禁用轻拍的卡牌
        self.tileFlipped = -1 //将翻转的按钮设置为-1，表示没有进行翻转的卡牌
    }else{
        //翻转第一个卡牌
        self.tileFlipped=senderID
        self.tile1=sender as UIButton
        var tileImage:UIImage=self.tiles.objectAtIndex(senderID) as UIImage
        sender.setImage(tileImage, forState: UIControlState.Normal)
    }
}
```

5.5.2 判断卡牌

当翻转两个卡牌后，就可以对这两个卡牌进行判断了。首先，需要添加一个变量，此变量是用来保存两个卡牌是否相等的值。代码如下：

```
var isMatch=false
```

然后需要添加判断的代码。因为判断卡牌是否相等是在翻转第二个卡牌后进行的，所以需要在 `tileClicked()` 方法中的禁用轻拍卡牌代码的后面添加以下代码：

```

sender.setImage(tileImage, forState: UIControlState.Normal)
isDisabled=true
//判断两个卡牌是否相等
if(tileImage==lastImage){
    //将两个按钮设置为不可用状态
    self.tile1.enabled=false
    self.tile2.enabled=false
    isMatch=true
}

```

5.5.3 配对成功和失败的操作

当两个卡牌相同或者不相同时，都应该有相应的操作。如果两个卡牌相同，就将这两个卡牌消除；如果两个卡牌不相同，就再次盖住，重新选择翻转的卡牌。为了便于检查玩家翻转卡牌，这里通过定时检查的方式进行判断。以下就是对此功能的实现，首先需要创建一个定时器，此定时器可以实现每隔一段时间后就会执行卡牌配对成功或者失败的操作。定时器的创建语法形式如下：

```

NSTimer. scheduledTimerWithTimeInterval(_ seconds: NSTimeInterval,
                                         target target: AnyObject,
                                         selector aSelector: Selector,
                                         userInfo userInfo: AnyObject?,
                                         repeats repeats: Bool
                                         )

```

其中，参数说明如下。

- ❑ `_seconds`: 用来指定两次触发所间隔的秒数；
- ❑ `target`: 用来指定消息发送的对象；
- ❑ `aSelector`: 用来指定触发器所调用的方法；
- ❑ `userInfo`: 该参数可以设定为 `nil`，当定时器失效时，由玩家指定的对象保留和释放该定时器；
- ❑ `repeats`: 用来指定是否重复调用自身。

创建一个 `NSTimer` 定时器的代码如下：

```

NSTimer.scheduledTimerWithTimeInterval(1.0,
                                         target: self,
                                         selector:Selector("resetTiles"),
                                         userInfo: nil,
                                         repeats: false
                                         )

```

创建好定时器后，就需添加在定时器中所执行的方法 `resetTiles()` 方法。此方法需要实现的功能是判断 `isMatch` 的值是否为 `true`，从而执行相应的操作。`isMatch` 中保存了两个卡牌是否相同的结果。代码如下：

```

func resetTiles(){
    //判定
    if(isMatch){
        //设置两个按钮的图像
        self.tile1.setImage(self.backTileImage, forState: UIControlState.Normal)
        self.tile2.setImage(self.backTileImage, forState: UIControlState.Normal)
    }else{

```



```
self.tile1.setImage(self.blankTileImage, forState: UIControlState.Normal)
self.tile2.setImage(self.blankTileImage, forState: UIControlState.Normal)
}
isDisabled=false
isMatch=false
}
```

此时运行程序，会看到如图 5.28 所示的运行效果。当翻转的两个卡牌相同时，就会进行自动消除。

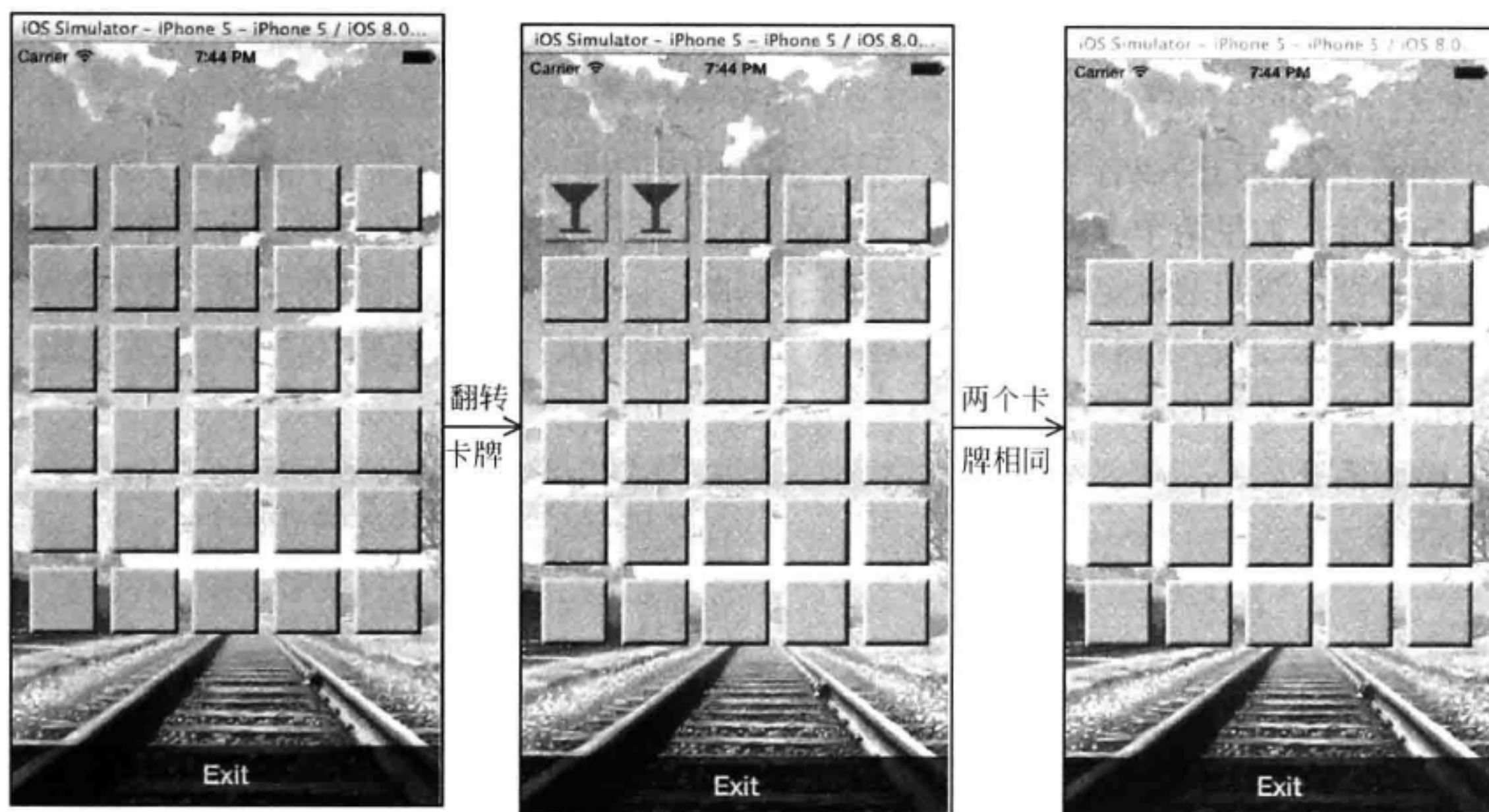


图 5.28 运行效果

当翻转的两个卡牌不相同时，就会再次反扣回去，如图 5.29 所示。

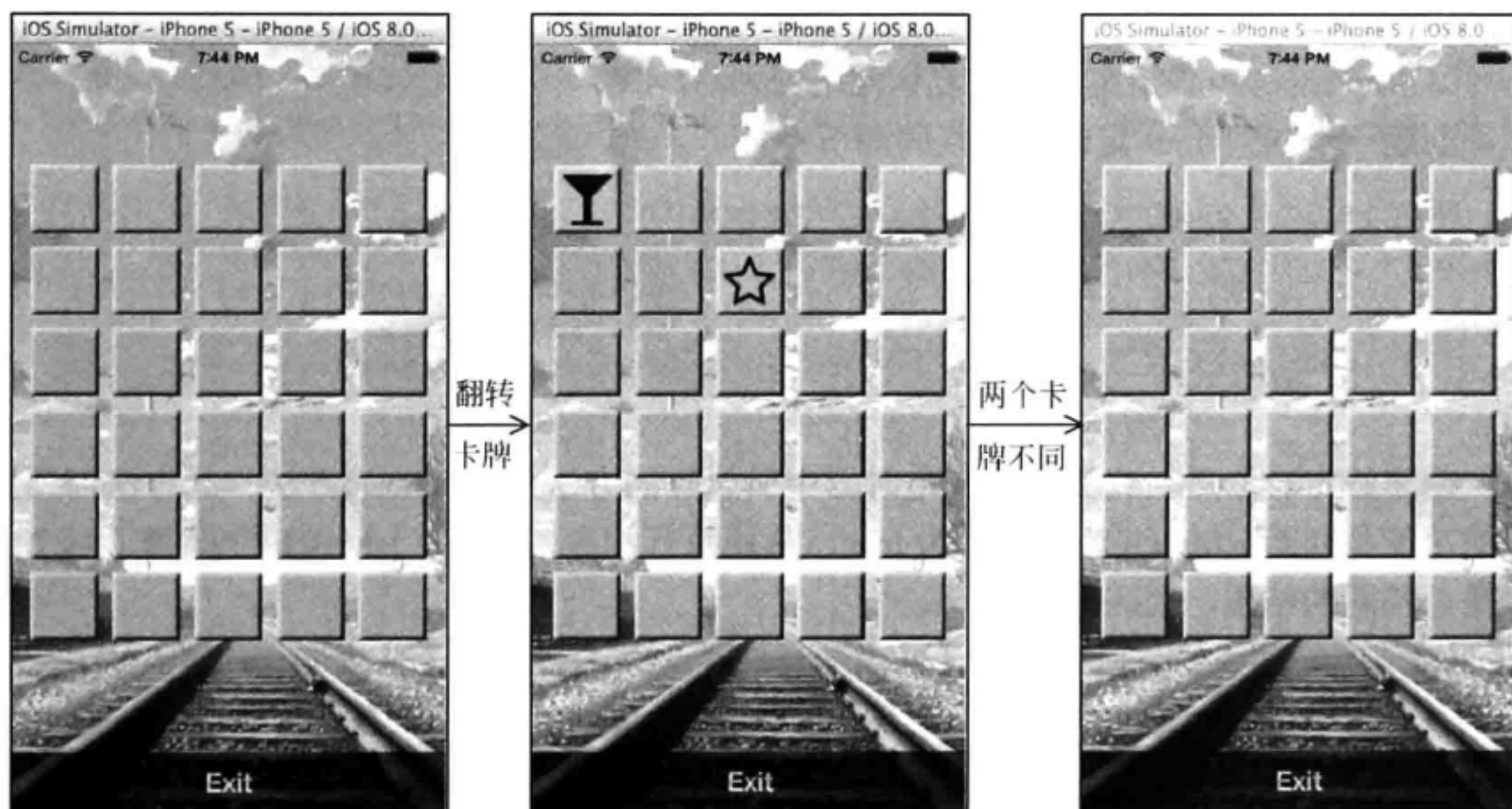


图 5.29 运行效果

5.5.4 完成所有配对

当配对界面的所有卡牌都配对完成后，就需要有一个提示。在此程序中，我们会在游戏结束后，显示一个标签对象，在标签对象中显示玩家完成游戏所用的猜测次数。以下是对此功能的实现。

1. 插座变量的关联与声明

由于需要在标签中显示内容，所以需要有一个标签对象，并且实现标签对象和插座变量的关联。在前面的章节讲解过插座变量的声明关联，它们是一起进行的。除此以外，插座变量的声明和关联也可以分开进行。首先，需要使用 `IBOutlet` 关键字对标签的插座变量进行声明，其代码如图 5.30 所示。

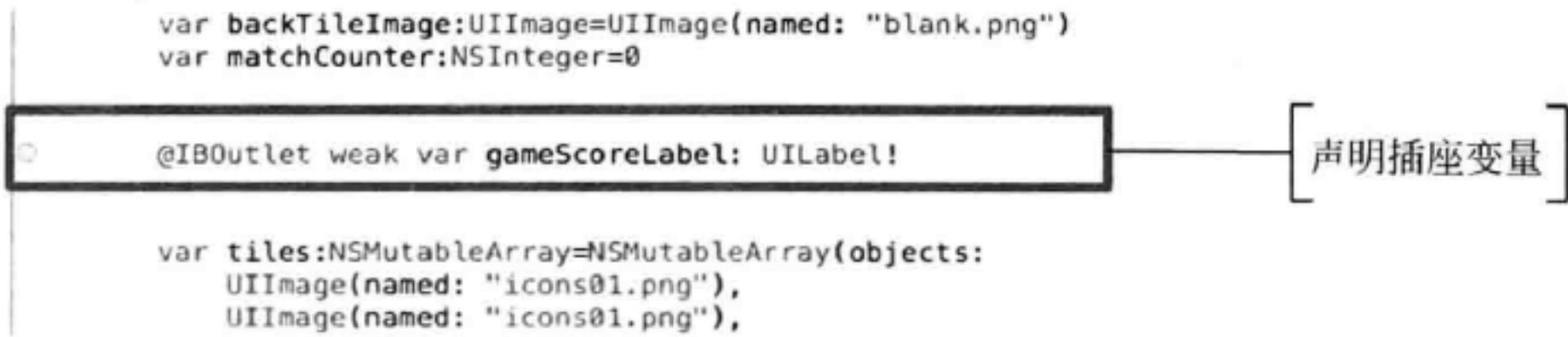



图 5.30 插座变量

 **注意：**声明好的插座变量会在代码的前面出现一个空心的小圆圈。此小圆圈表示该插座变量还未进行关联。

如何将此插座变量和 `GameScene View Controller` 视图控制器的界面中的标签对象进行关联就是以下所要讲解的内容。具体的操作步骤如下所述。

(1) 使用调整窗口中的工具，将 Xcode 的界面进行调整。将其调整为和图 5.18 一样的效果。

(2) 按住 `Ctrl` 键拖动界面中的标签对象，这时会出现一个蓝色的线条，将这个蓝色的线条和文件 `Game.Swift` 中的插座变量进行关联，如图 5.31 所示。

(3) 松开鼠标后，标签对象就与插座变量成功的关联在一起了，此时插座变量前面的空心小圆圈就变为了实心的小圆圈，它表示此插座变量已被关联，如图 5.32 所示。

2. 完成配对，显示信息

插座变量关联好后，需要声明定义两个变量，代码如下：

```
var matchCounter:NSInteger=0
var guessCounter:NSInteger=0
```

其中，`matchCounter` 变量用来保存现在消失的卡牌对数；`guessCounter` 变量用来保存玩家猜测的次数。当玩家每配对成功一次，就需要将 `matchCounter` 变量加 1，所以需要在 `tileClicked()` 方法的判断两个卡牌是否相等的地方添加代码，代码如下：

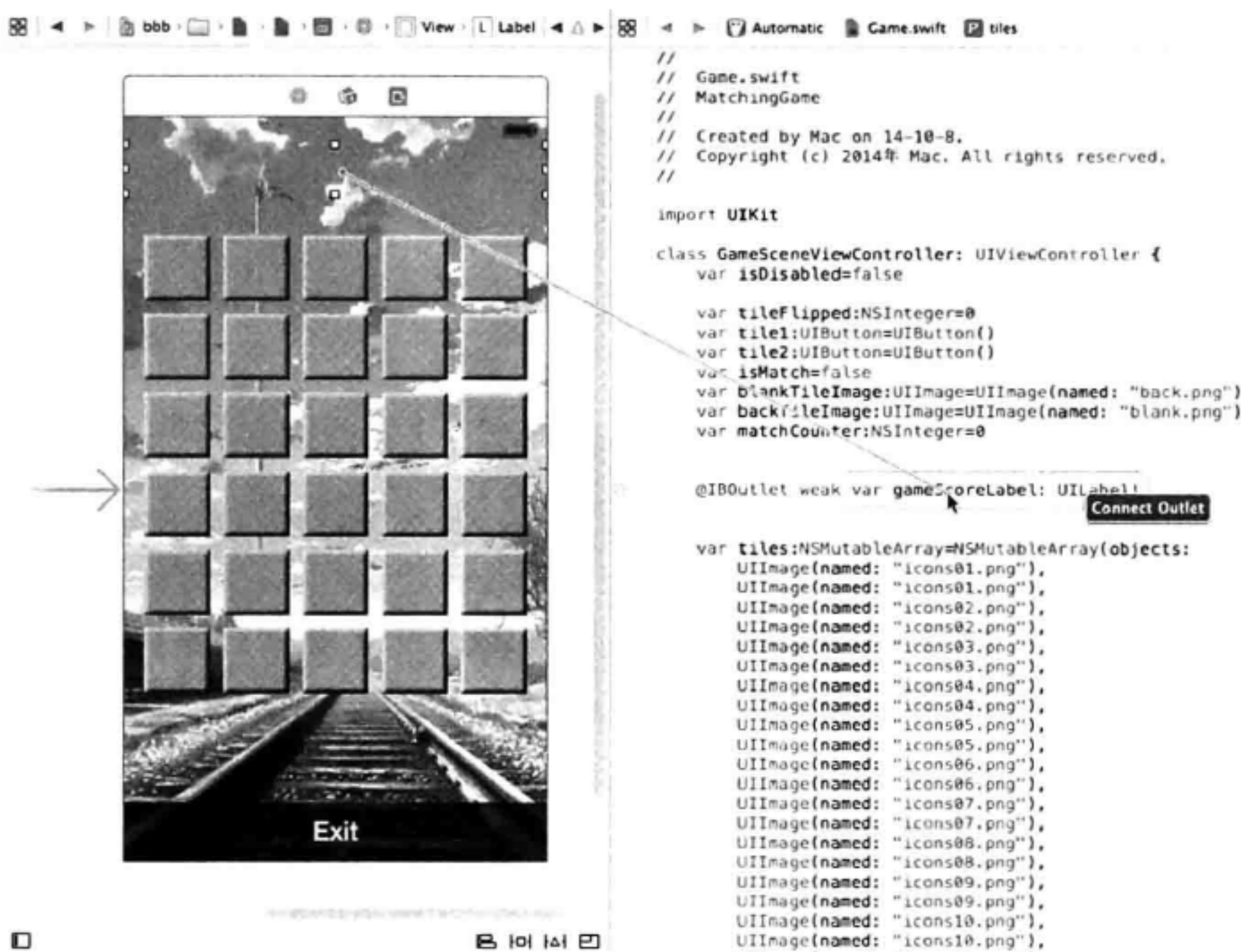


图 5.31 操作步骤

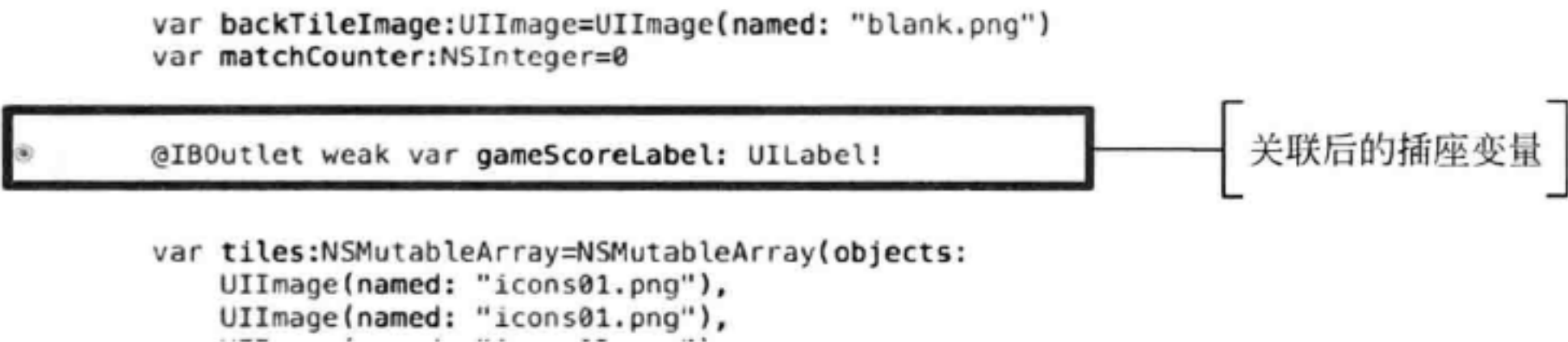


图 5.32 关联后的插座变量

```
if (tileImage==lastImage) {
    self.tile1.enabled=false
    self.tile2.enabled=false
    isMatch=true
    self.matchCounter++
}
```

当 matchCounter 变量中保存的数值和 self.tiles.count/2 数值一致时，表示在配对界面中的卡牌已全部配对完成，所以需要在 resetTiles()方法中添加以下代码：

```
isMatch=false
if (self.matchCounter==(self.tiles.count/2)) {
    self.winner()
}
```

在代码中调用了 winner()方法，此方法就是在标签中显示玩家猜测的次数，所以需要添加 winner()方法。代码如下：

```
func winner() {
    self.gameScoreLabel.text="You won with \(self.guessCounter) Guesses";
}
```


此时运行程序，会看到如图 5.33 所示的效果。

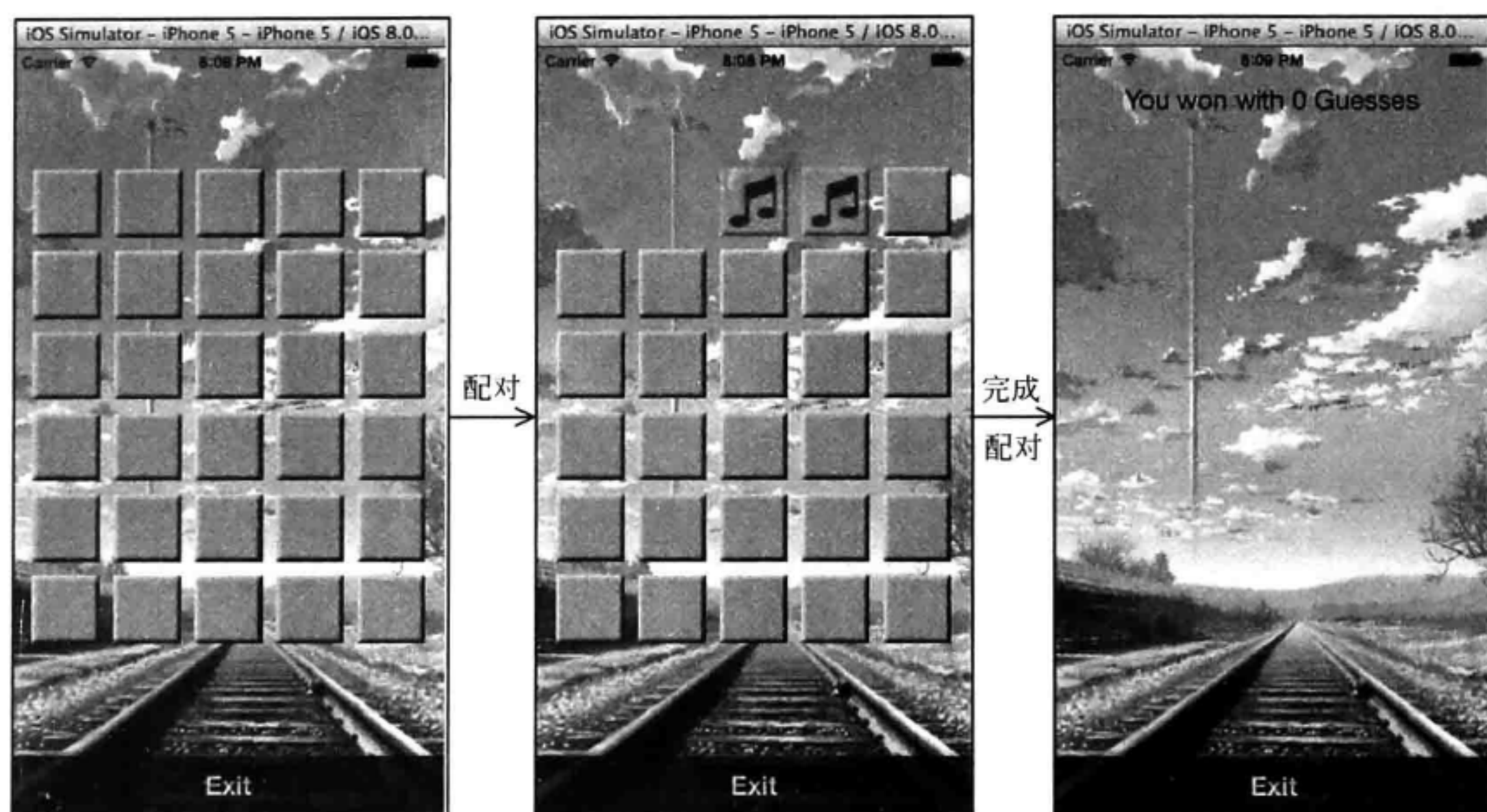


图 5.33 运行效果

5.6 配对模块的额外功能

本节将讲解记忆配对游戏的一些额外的功能，如猜测次数记录功能，以及调整游戏难度功能。

5.6.1 猜测次数功能

在图 5.33 中我们会看到，当配对游戏完成后，显示在标签中的内容是 You won with 0 Guesses，此时还没有实现计算玩家猜测次数的功能，以下是对此功能的具体实现。当玩家每翻转两个卡牌后，不管卡牌是否相同，都是玩家进行的猜测，所以需要在 `tileClicked()` 方法中添加以下的代码：

```
sender.setImage(tileImage, forState: UIControlState.Normal)
isDisabled=true
self.guessCounter++
```

此代码实现的功能就是计算玩家猜测的次数，如果一直显示猜测的次数，并非在游戏结束后，还需要在 `tileClicked()` 方法的结尾处添加以下的代码：

```
gameScoreLabel.text="Matches: \(self.matchCounter),Guesses:\(self.guessCounter)"
```

此时运行程序，可以看到如图 5.34 所示的效果。

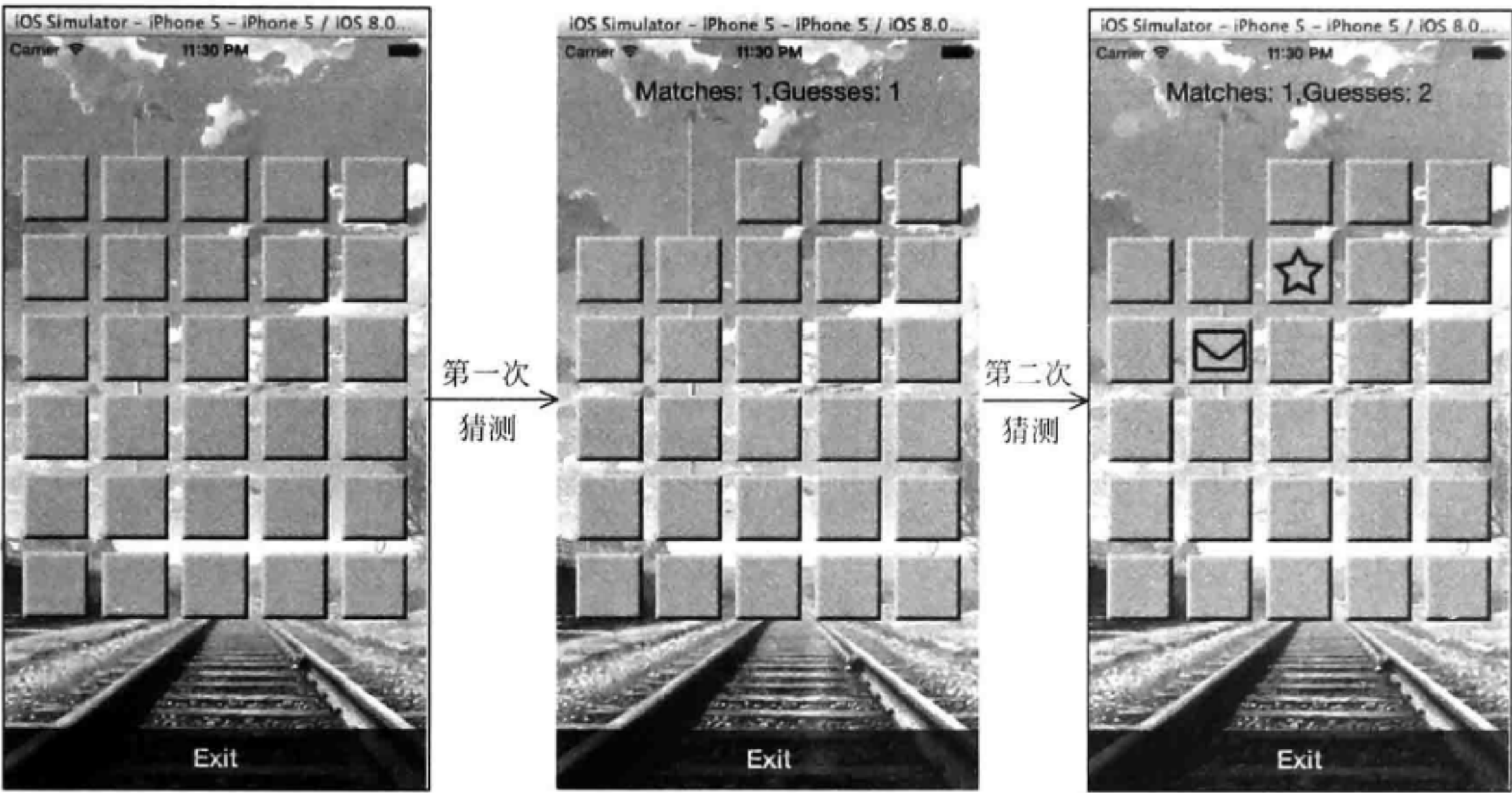


图 5.34 运行效果

⚠注意：在运行程序时，当轻拍按钮后，才会在标签中显示内容。如果想要在程序一开始就显示内容，需要在 Game.swift 文件中添加 viewDidLoad()方法。此方法会在程序加载后自动调用。代码如下：

```
override func viewDidLoad() {
    super.viewDidLoad()
    gameScoreLabel.text="Matches:\(self.matchCounter) ,Guesses:\(self.
    guessCounter) "
}
```

5.6.2 提高游戏的难度

现在轻拍后的卡牌和在数组对象 tiles 中保存的图像顺序是一致的，没有随机性，导致每次游戏重新开始后的卡牌顺序是不变的。这样一来，游戏就变得太简单了，没有趣味性。如果开发者想要提高游戏的难度，就需要使用游戏中卡牌的随机性，即产生的随机数。在 Swift 中，随机数的产生一般可以使用 arc4random()方法实现。以下是提供游戏难度的具体做法。

首先实例化一个 shuffledTiles 属性对象，代码如下：

```
var shuffledTiles:NSMutableArray=NSMutableArray()
```

然后，添加一个 shuffleTiles()方法，让翻转后的卡牌随机生成。代码如下：

```
func shuffleTiles() {
    var tileCount:Int=tiles.count
    var tileID:Int=0
    //为数组添加对象
```

```

for (tileID; tileID < (tileCount/2); tileID++) {
    self.shuffledTiles.addObject (NSNumber numberWithInt (Int32 (tileID)))
    self.shuffledTiles.addObject (NSNumber numberWithInt (Int32 (tileID)))
}
var i:UInt32=0
for (i; i < UInt32 (tileCount); ++i) {
    var nElements:NSInteger=UInt32 (tileCount)-i
    var a=arc4random() //产生随机数
    var b=a%UInt32 (nElements)
    var n=b+i
    //交换对象
    self.shuffledTiles.exchangeObjectAtIndex (Int (i), withObjectAtIndex:
    Int (n))
    self.tiles.exchangeObjectAtIndex (Int (i), withObjectAtIndex:
    Int (n))
}
}

```

最后，需要在 viewDidLoad() 方法中添加调用 shuffleTiles() 方法的代码。代码如下：

```
self.shuffleTiles()
```

此时运行程序，可以看到如图 5.35 所示的效果。

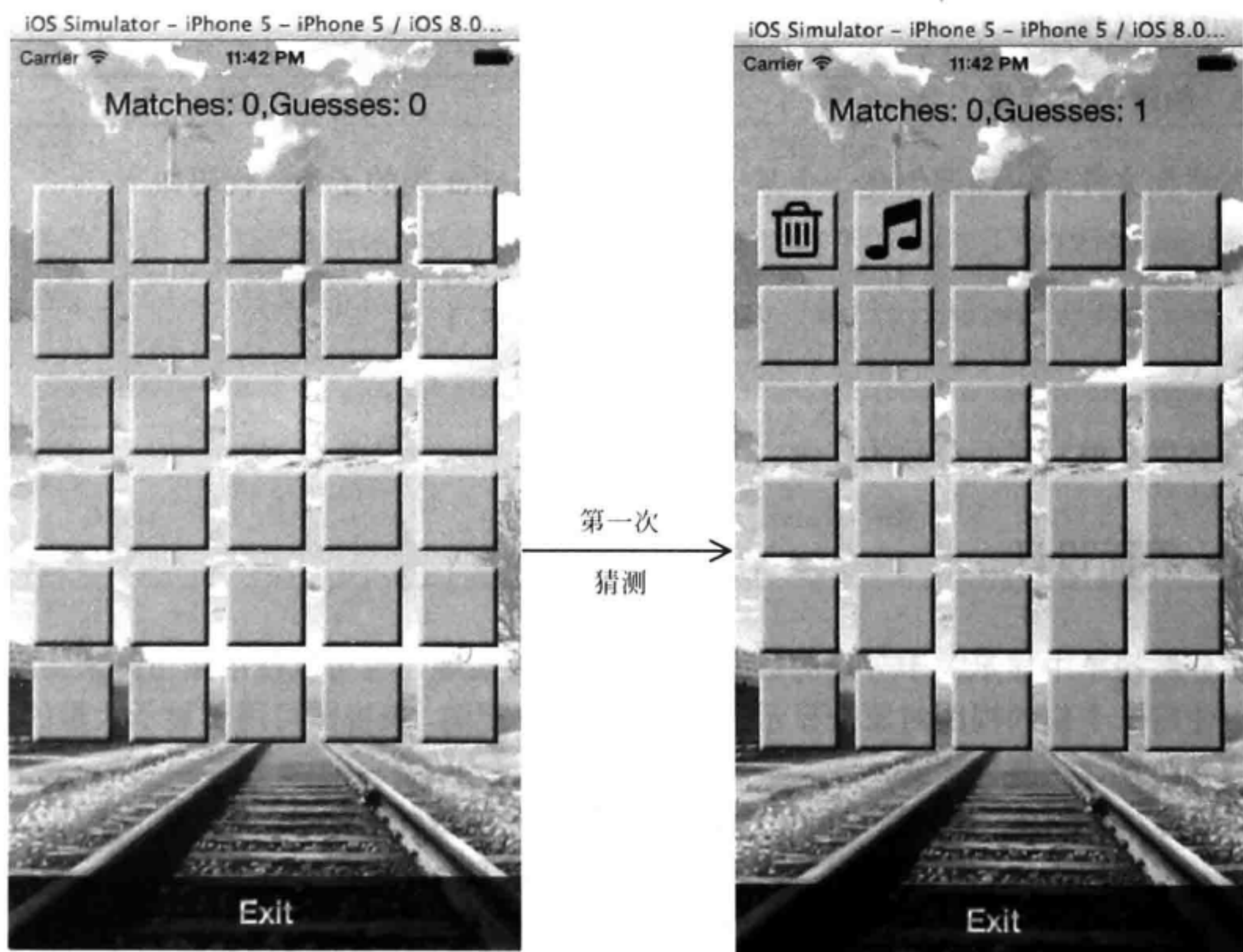


图 5.35 运行效果

5.7 分数榜单模块

在很多的比拼游戏中都会有一个关于分数的排行榜。在这个分数的排行榜中会显示一个关于分数的数据。本节将讲解分数榜单的设计，以及实现显示分数的效果。

5.7.1 准备工作

在设计分数排行榜之前，首先需要做一些准备工作，如文件的创建等。

1. 创建文件

开发者需要创建一个 Swift File 模板类型的文件，命名为 List.swift，具体操作步骤如下所述。

(1) 选择菜单栏上的 File|New|File 命令，弹出 Choose a template for your new file:对话框。

(2) 选择 iOS|Source|Swift file 模板，然后单击 Next 按钮，弹出设置文件信息的对话框，其中包括文件名称和位置。

(3) 输入文件名称 List 后，单击 Create 按钮，此时就在创建的项目中新建了一个 List.swift 的文件。

2. 创建一个空类

在添加文件后，需要创建一个基于 UIViewController 类的子类。代码如下：

```
import UIKit

class ListViewController: UIViewController {

}
```

该类用于编写显示分数的代码。

5.7.2 界面设计

在分数榜单中显示关于分数的数据可以使用标签实现，也可以使用按钮等来实现。本小节中使用一个新的视图对象来显示分数，即使用表视图。表视图可用于显示大量的数据。以下是界面设计的具体步骤。

(1) 在视图对象库中拖动 View Controller 视图控制器对象到画布中。

(2) 单击新添加的视图控制器，选择界面上方的 Dock 中的 View Controller 图标。在工具窗口中的 Show the Identity inspector 选项，即属性查看器，将 Custom Class 下的 Class 设置为创建的 ListViewController 类。这时在画布中的这个视图控制器就变为了 List View Controller 视图控制器。

(3) 对 List View Controller 视图控制器的界面进行设计，效果如图 5.36 所示。

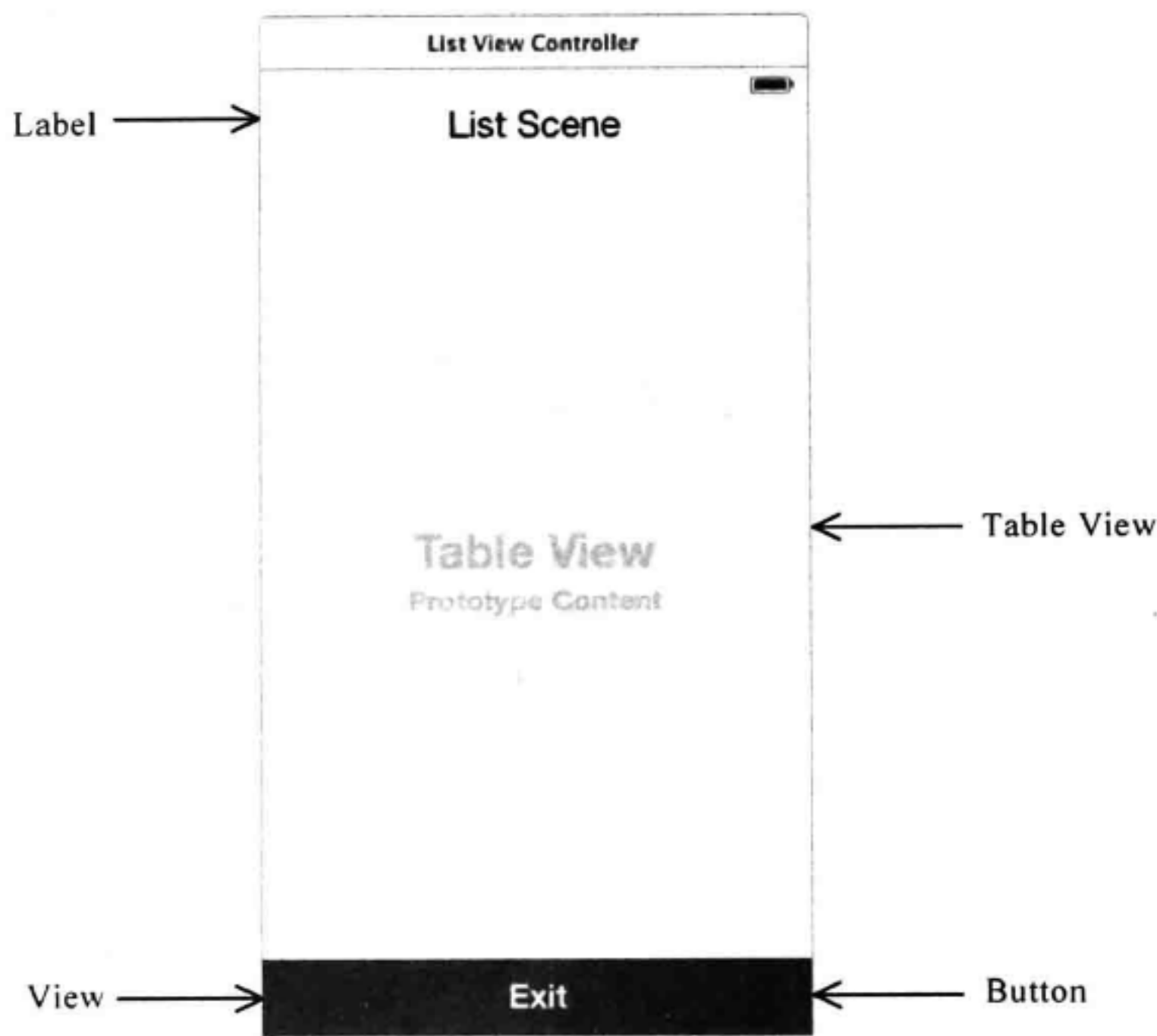


图 5.36 界面的效果

需要添加的视图对象，以及对它们的设置，如表 5-4 所示。

表 5-4 设置界面

视 图	设 置
Label	Text: List Scene Font: System 22.0 Alignment: 居中 位置和大小: (89, 20, 143, 25)
Table View	与 dataSource 关联 与 delegate 关联 位置和大小: (0, 53, 320, 461)
View	Alpha: 0.65 Background: 黑色 位置和大小: (0, 523, 320, 45)
Button	Title: Exit Font: System Bold 20.0 Text: 白色 位置和大小: (78, 7, 165, 30)

需要注意的是，图 5.36 中的表视图是从视图对象库中拖动到界面中的。对于 dataSource 与 delegate 的关联步骤如下所述。

(1) 右击界面中的 Table View 表视图对象，会弹出一个 Table View 对话框，如图 5.37 所示。

(2) 按住 Ctrl 键拖动 dataSource 到 Dock 中的 List View Controller 图标上，如图 5.38 所示。

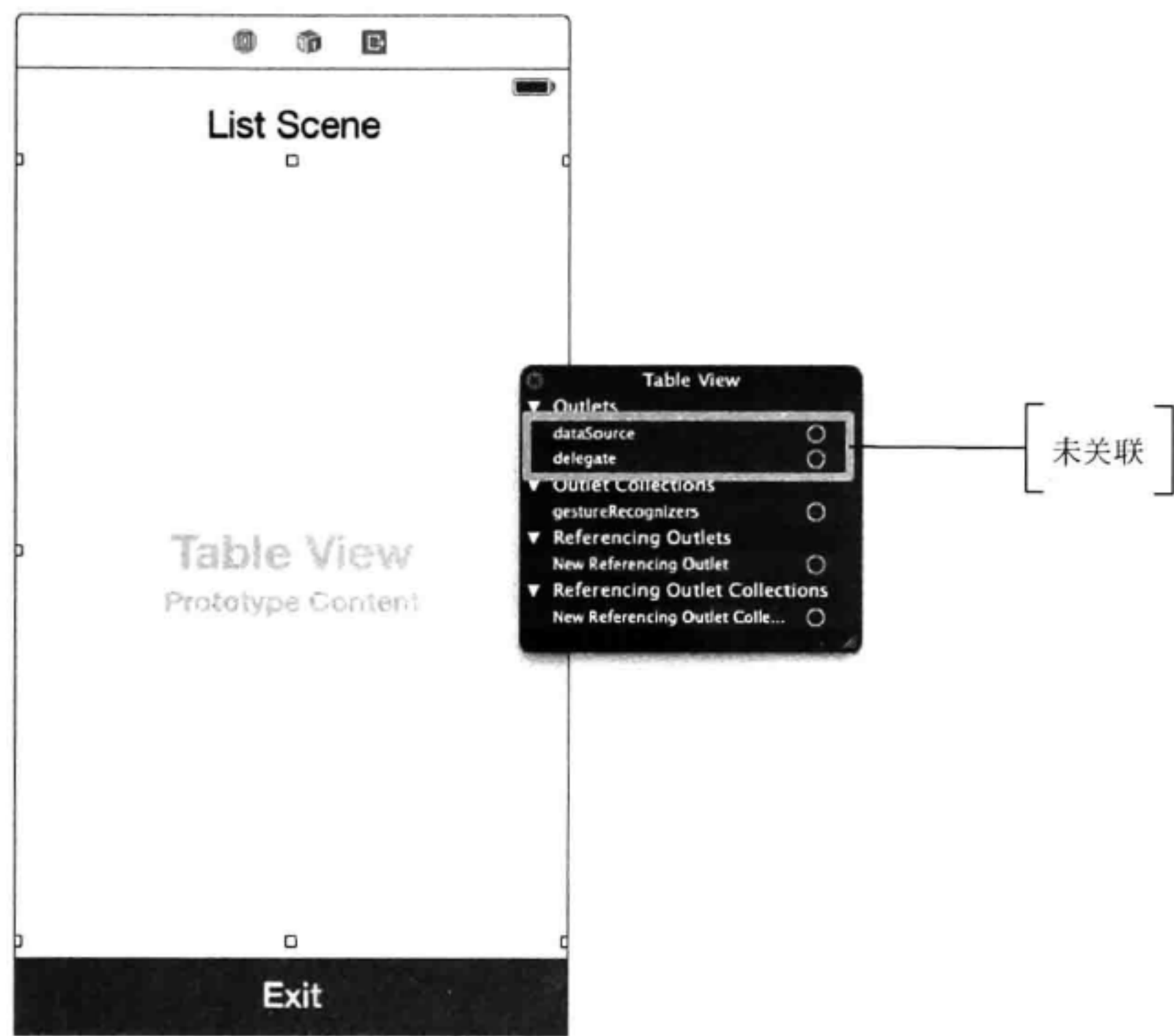


图 5.37 操作步骤 1

(3) 松开鼠标后，dataSource 就实现了关联。此时 dataSource 与 List View Controller 进行了关联，如图 5.39 所示。

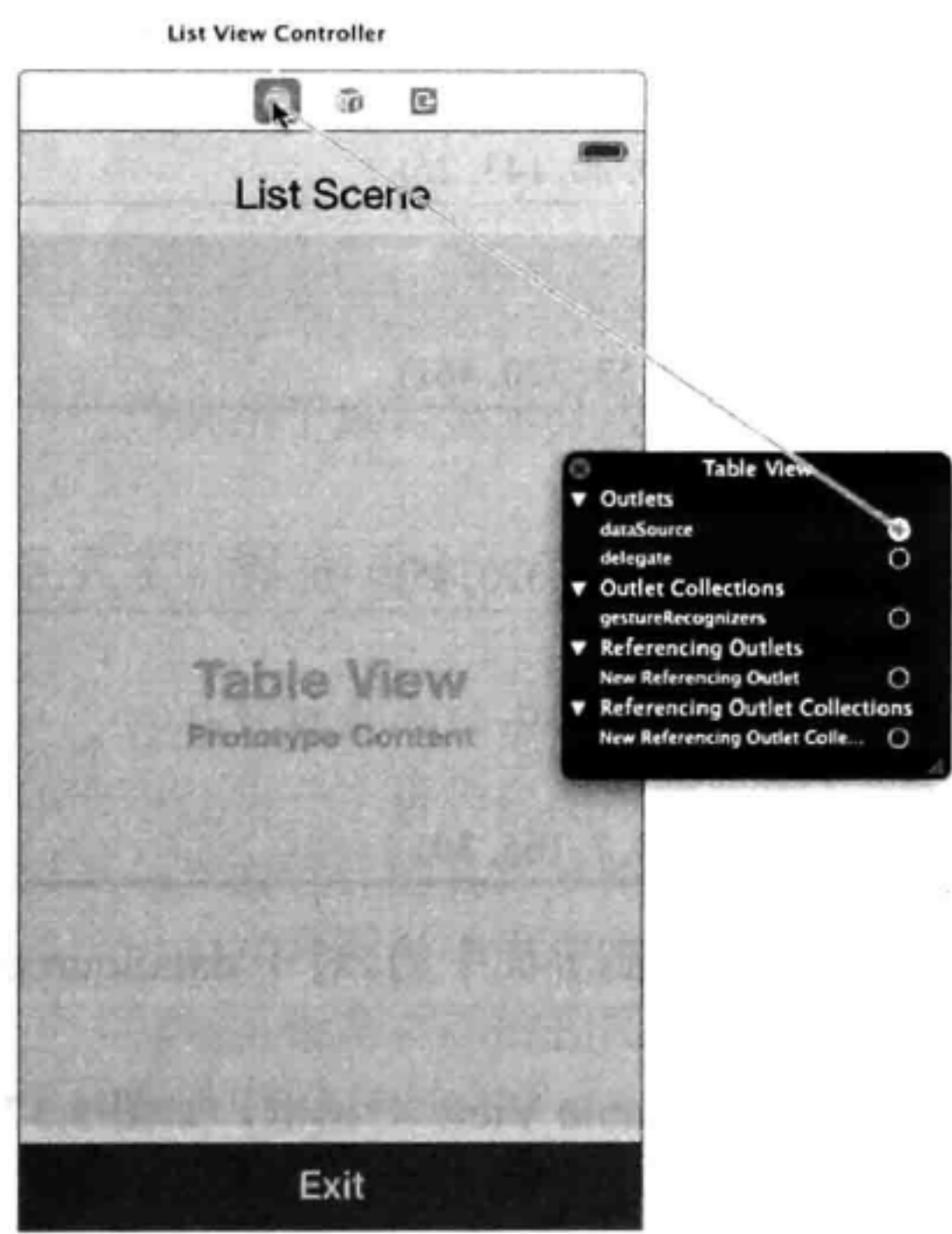


图 5.38 操作步骤 2

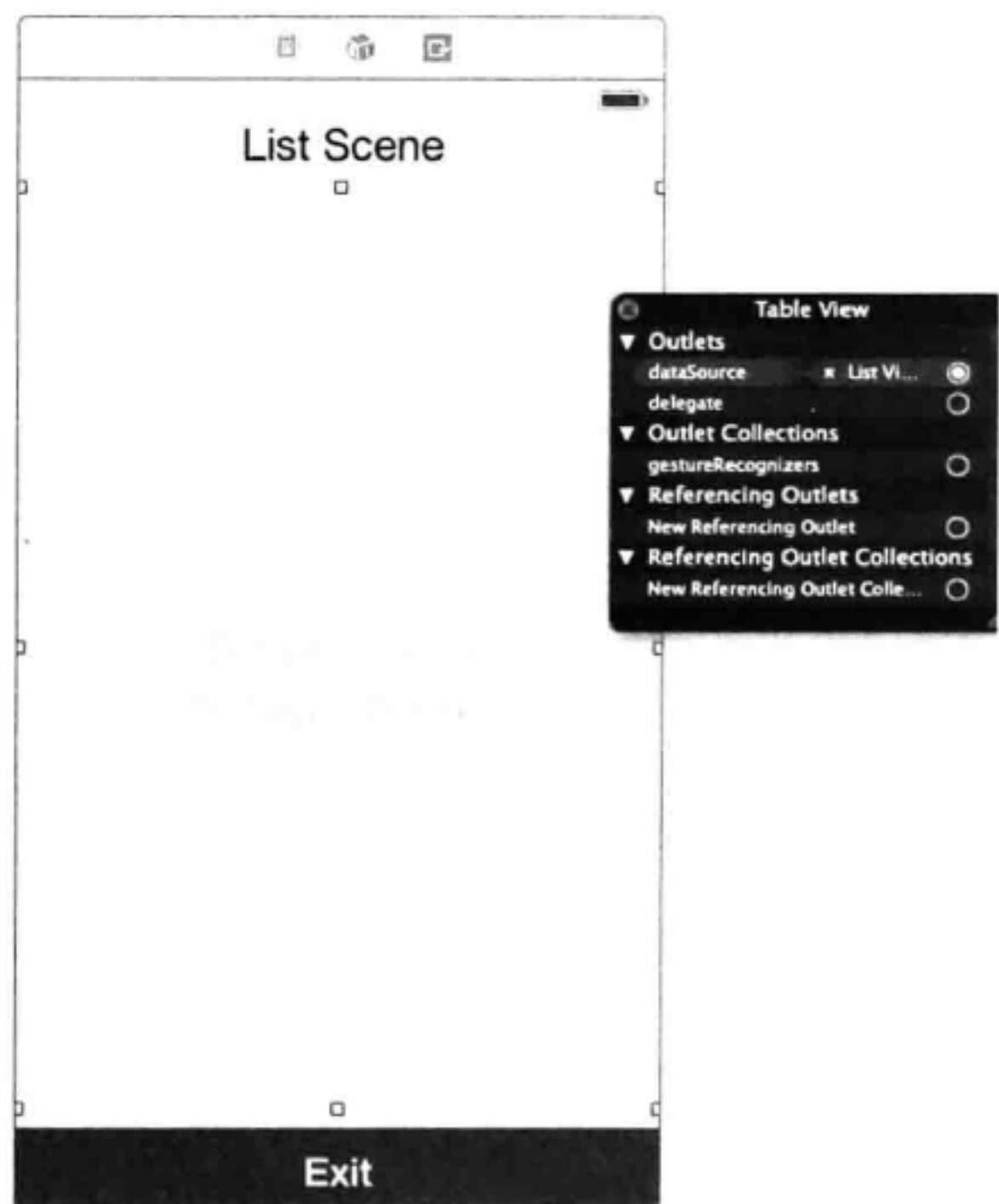


图 5.39 操作步骤 3

🔔注意: delegate 的关联步骤和 dataSource 的关联步骤是一样的。

5.7.3 实现分数的显示

要显示的分数其实就是玩家在完成游戏后总的猜测次数。首先, 需要创建一个可变的数组对象, 在此数组对象中保存玩家的猜测次数。可变数组对象的创建代码如下:

```
var score:NSMutableArray=NSMutableArray()
```

然后在玩家完成配对游戏后, 弹出一个对话框, 询问玩家是否保存此次的游戏结果。对于弹出的对话框, 可以使用 UIAlertView 警告视图对象进行实现。警告视图对象的功能是把需要注意的信息显示给玩家。一般显示一条信息, 或者显示一条信息和一个按钮。警告视图对象的创建语法形式如下:

```
UIAlertView(title title: String?,
            message message: String?,
            delegate delegate: AnyObject?,
            cancelButtonTitle cancelButtonTitle: String?
            )
```

其中, 参数说明如下。

- ❑ title: 用来显示初始化并设置出现在警告视图顶端的标题;
- ❑ message: 用来指定将出现在对话框内容区域的字符串;
- ❑ delegate: 用来指定将充当提醒委托的对象(所谓委托顾名思义就是委托别人办事, 就是当一件事情发生后, 自己不处理, 让别人来处理);
- ❑ cancelButtonTitle: 用来指定警告视图中默认按钮的标题。

一般警告视图的形式如图 5.40 所示。

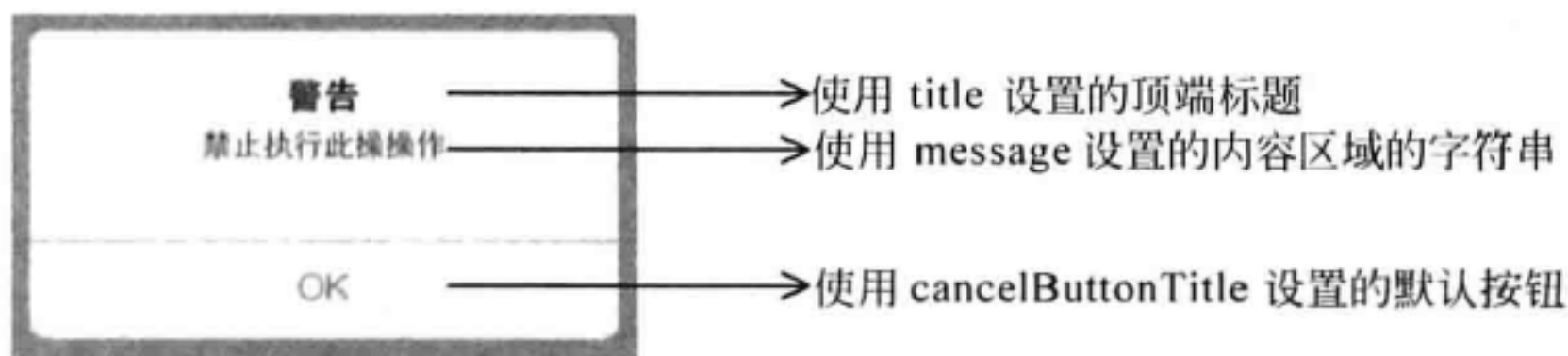


图 5.40 警告视图

在 winner()方法中显示警告视图的代码。代码如下:

```
self.gameScoreLabel.text="You won with \(self.guessCounter) Guesses";
//创建警告视图对象
var alert=UIAlertView(title: "游戏已完成",
                      message: "是否保存猜测数字",
                      delegate:self,
                      cancelButtonTitle: "否"
                      )
alert.addButtonWithTitle("是") //添加按钮
alert.show() //显示警告视图对象
```

此时创建的警告视图的按钮是没有响应的, 可以添加以下代码实现按钮的响应, 即实

现在轻拍警告视图中的“是”按钮时，将玩家的猜测次数保存在可变数组对象 `score` 中。代码如下：

```
func alertView(_alertView: UIAlertView, clickedButtonAtIndex buttonIndex: Int)
{
    var name: NSString = _alertView.buttonTitleAtIndex(buttonIndex)
    //判断当前轻拍按钮的标题是否为“是”
    if(name.isEqualToString("是")){
        score.addObject(self.gameScoreLabel.text!) //为数组添加对象
    }
}
```

打开 `List.swift` 文件，在此文件中实现将玩家保存的猜测次数显示在分数榜单的表视图中，即在表视图中显示字符串。要想在表视图中显示字符串，必须要实现以下 3 个步骤。

1. 设置表视图的节数

所谓节数，是对于分组表所说的，意思是在分组表中要分为几组，对应的每一组就是一个节，如图 5.41 所示。

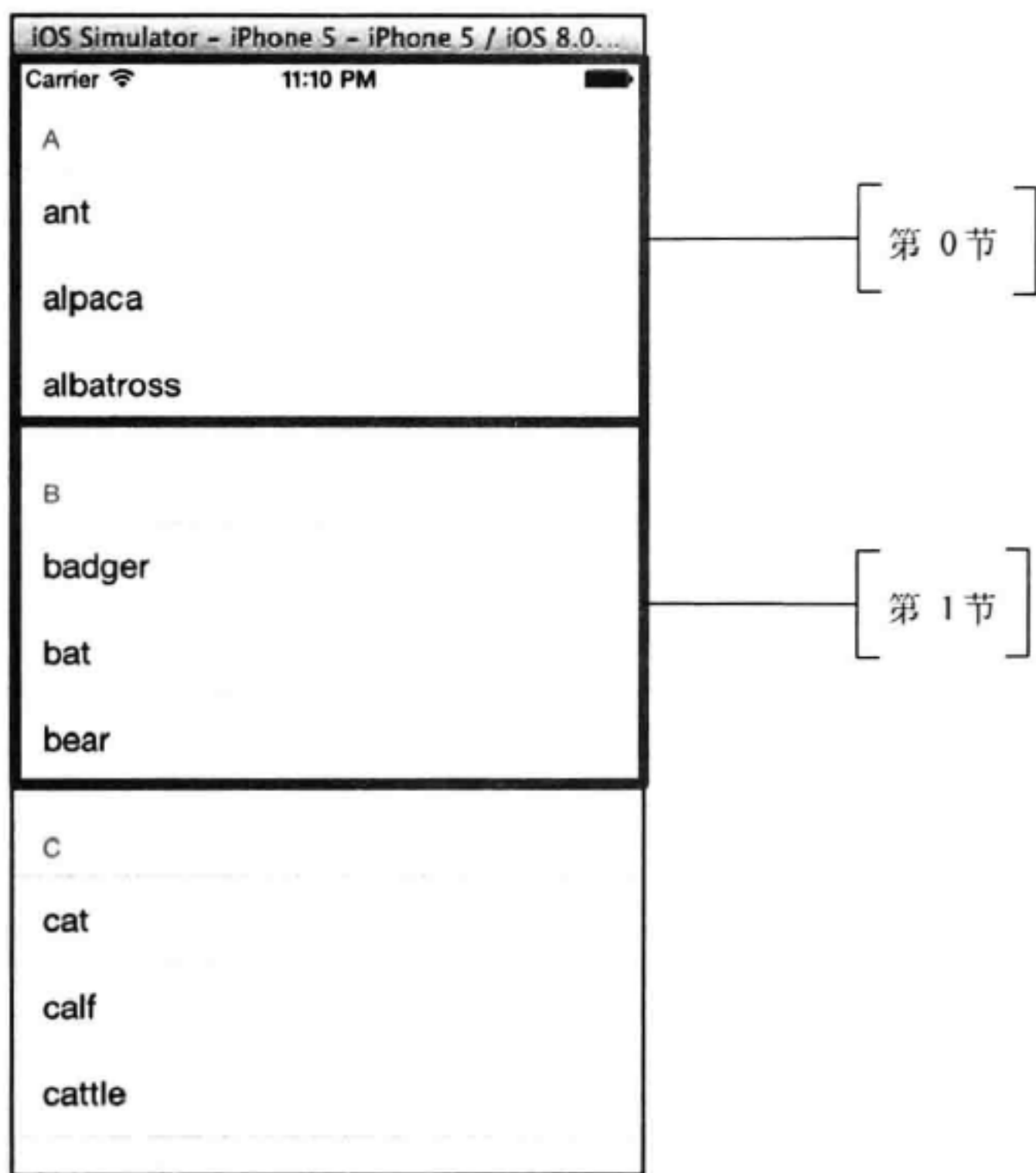


图 5.41 分组表

注意：在一个分组表中，最开始的节被称为第 0 节。

一般要设置节数，需要使用 `numberOfSectionsInTableView()` 方法，其语法形式如下：

```
func numberOfSectionsInTableView(_ tableView: UITableView) -> Int
```

注意：一般不对此方法进行设置，默认为只有 1 节，也就是不分组的表视图。

2. 设置表视图的行数

知道了表视图的节数后，就可以来设置在表视图要填充的行数，一般使用方法 `tableView(tableView: UITableView, numberOfRowsInSection section: Int)` 对表视图的行数进行设置。其语法形式如下：

```
func tableView(tableView: UITableView, numberOfRowsInSection section: Int)
-> Int
```

其中，`tableView` 用来指定表视图；`section` 用来指定索引号，此索引号实现的功能是识别表视图中的节。

3. 插入表单元

在将表视图的节数和行数都设置好之后，就是来插入表单元了。`UITableViewCell` 称为表单元。在表视图中，每一行都是一个表单元的实例。所有的表单元就构成了一个表。如果没有表单元，那么表就是一个空白表。要实现在表视图特定的位置插入一个表单元就要使用 `tableView(tableView: UITableView, cellForRowAt indexPath: NSIndexPath)` 方法。其语法形式如下：

```
func tableView(tableView: UITableView, cellForRowAt indexPath: NSIndexPath)
->UITableViewCell
```

其中，`tableView` 用来指定表视图；`indexPath` 用来指定一个索引路径，指定在表视图中的行。在此方法中，需要注意，要插入表单元，首先要对其进行创建，其语法形式如下：

```
UITableViewCell(style style: UITableViewCellStyle, reuseIdentifier
reuseIdentifier: String?)
```

其中，`style` 用来指定表单元的显示风格，这些风格有 `UITableViewCellStyleDefault`、`UITableViewCellStyleValue1`、`UITableViewCellStyleValue2`、`UITableViewCellStyleSubtitle`。`reuseIdentifier` 是一个重用标识符，当为创建的表单元格指定重用标识符后，单元格就可以进行重用。当创建的单元格传递为 `nil` 时，表明单元格对象是不被重用的。在 `List.swift` 文件中添加的代码如下：

```
import UIKit
class ListViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
    }
    //返回表视图的节数
    func numberOfSectionsInTableView(tableView: UITableView) -> Int {
        return 1
    }
    //返回表视图的行数
    func tableView(tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
```




```

        return score.count
    }
    //返回表视图的表单元格
    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath:
    NSIndexPath) ->UITableViewCell{
        var CellIdentifier:NSString="Cell"
        var cell:UITableViewCell? = tableView.dequeueReusableCellWithIdentifier
        (CellIdentifier) as? UITableViewCell
        //判断表单元格是否为空
        if(cell == nil){
            cell=UITableViewCell(style: UITableViewCellStyle.Default,
            reuseIdentifier: CellIdentifier)
        }
        cell!.textLabel?.text="\ (score.objectAtIndex(indexPath.row)) "
        return cell!
    }
}

```

此时运行程序，当玩家进入分数榜单的界面时，可以看到如图 5.42 所示的效果。

注意：此时是没有猜测次数的。

当玩家回到游戏界面，进行卡牌的配对，当配对结束后，会弹出一个警告视图的对话框，如图 5.43 所示。此警告视图用来询问玩家是否需要将猜测的次数进行保存，如果轻拍警告视图中的“是”按钮，在返回到分数榜单的界面后，会看到如图 5.44 所示的效果。

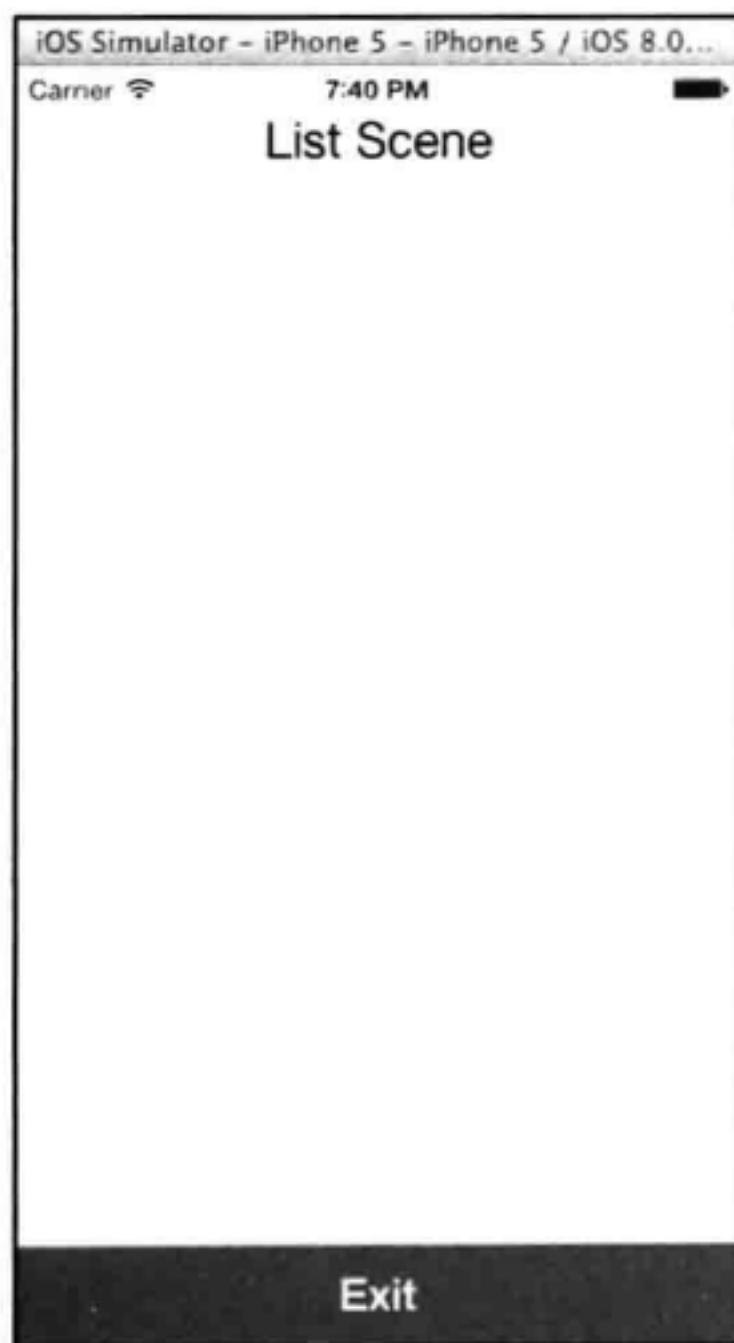


图 5.42 运行效果 1



图 5.43 运行效果 2

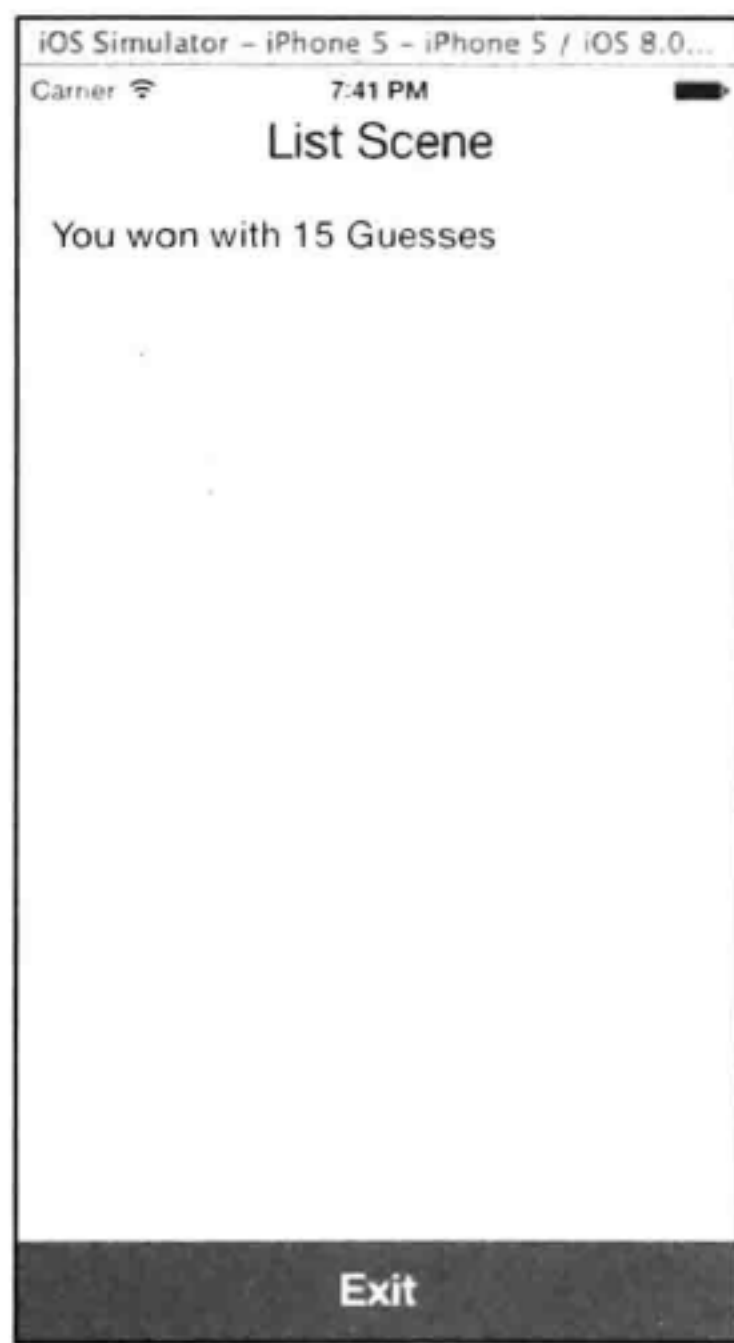


图 5.44 运行效果 3

⚠注意：在此运行界面中我们使用到了场景的切换，对于场景的切换，会在 5.9 节中进行讲解。

5.8 关于游戏模块

一般，游戏都有一个关于游戏的界面。在此界面中存放了游戏的介绍和玩法等内容。本节将讲解关于游戏这一模块的界面设计。在视图对象库中拖动 View Controller 视图控制器对象到画布中，然后对此视图控制器的界面进行设计，效果如图 5.45 所示。

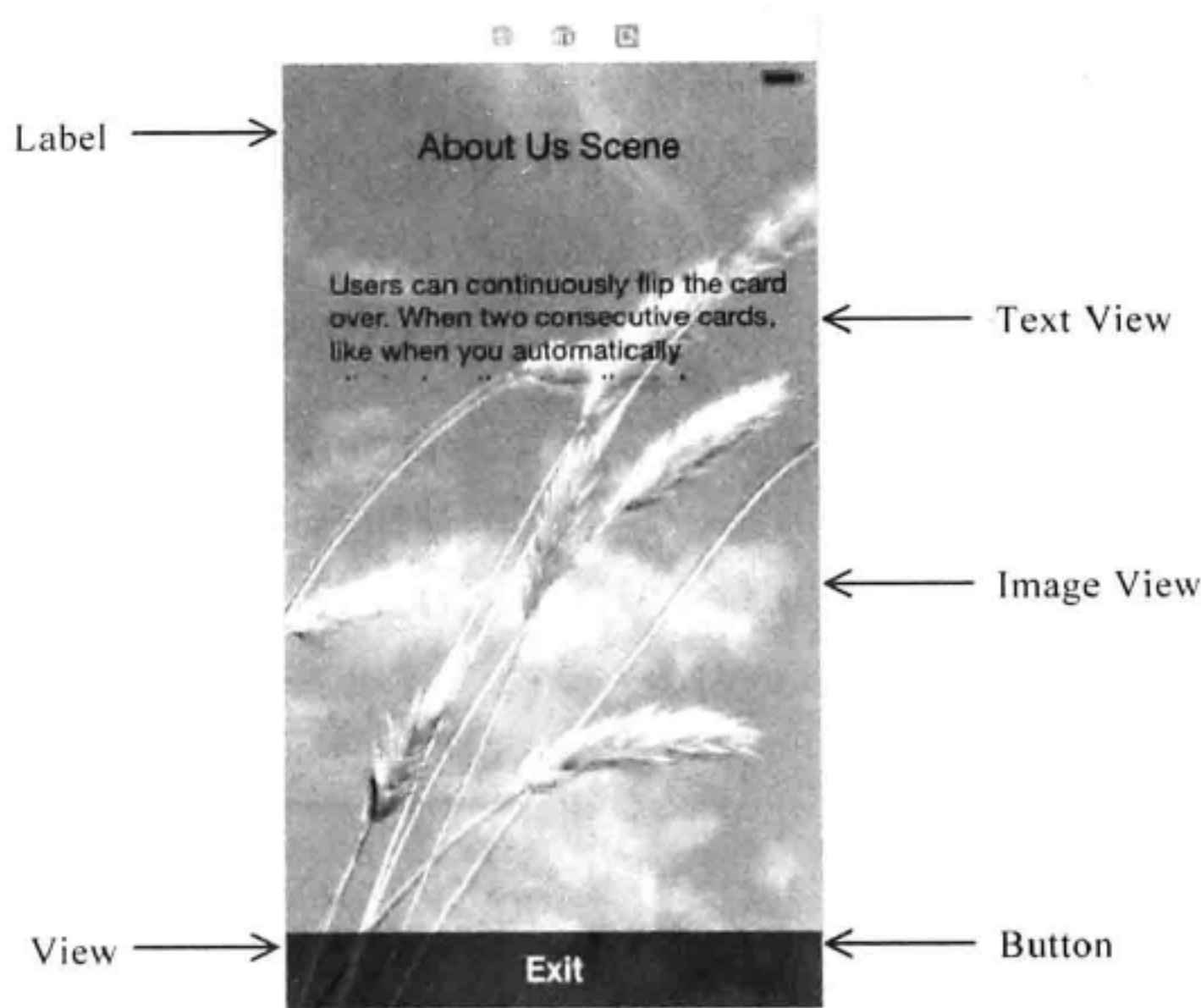


图 5.45 界面的效果

需要添加的视图对象，以及对它们的设置，如表 5-5 所示。

表 5-5 设置界面

视 图	设 置
Image View	Image: backdrop3.jpg 位置和大小: (0, 0, 320, 568)
Label	Text: About Us Scene Font: System 22.0 Alignment: 居中 位置和大小: (74, 38, 173, 25)
Text View	Text: Users can continuously flip the card over. When two consecutive cards, like when you automatically eliminate; otherwise, these two cards will be re-Flip back. Font: System 17.0 取消 Editable 复选框的选择 Background: 透明 位置和大小: (23, 114, 288, 312)

续表

视 图	设 置
View	Alpha: 0.65 Background: 黑色 位置和大小: (0, 523, 320, 45)
Button	Title: Exit Font: System Bold 20.0 Text: 白色 位置和大小: (78, 7, 165, 30)

5.9 场 景 切 换

完成以上模块后，下面我们需要将这些模块组合成一个完整的游戏。这就是场景切换。

5.9.1 什么是场景切换

在游戏中，每一个场景都不是单独存在的。玩家可以从一个场景中切换到另外一个场景中。本小节将讲解场景切换。在每一个游戏中都会使用到场景与场景的切换功能，例如，在街机原始人游戏中，选择关卡这一场景，可以看到有 4 关，这 4 关分别代表了 4 个场景，如图 5.46 所示。

当玩家选择第一关时，就会由选择关卡的场景跳转到第一关的场景中，如图 5.47 所示。

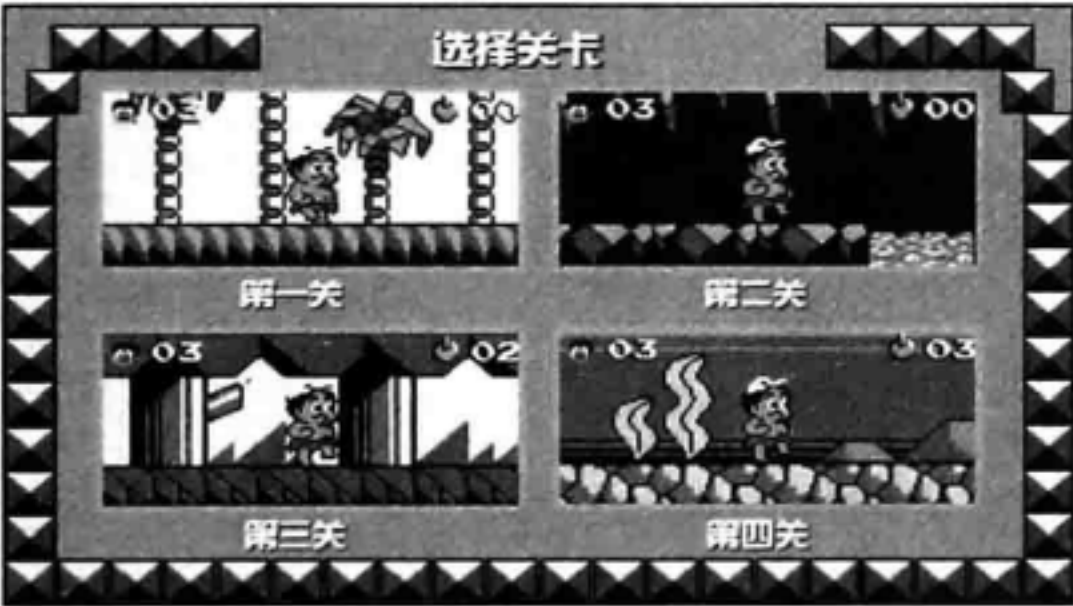


图 5.46 街机原始的关卡选择

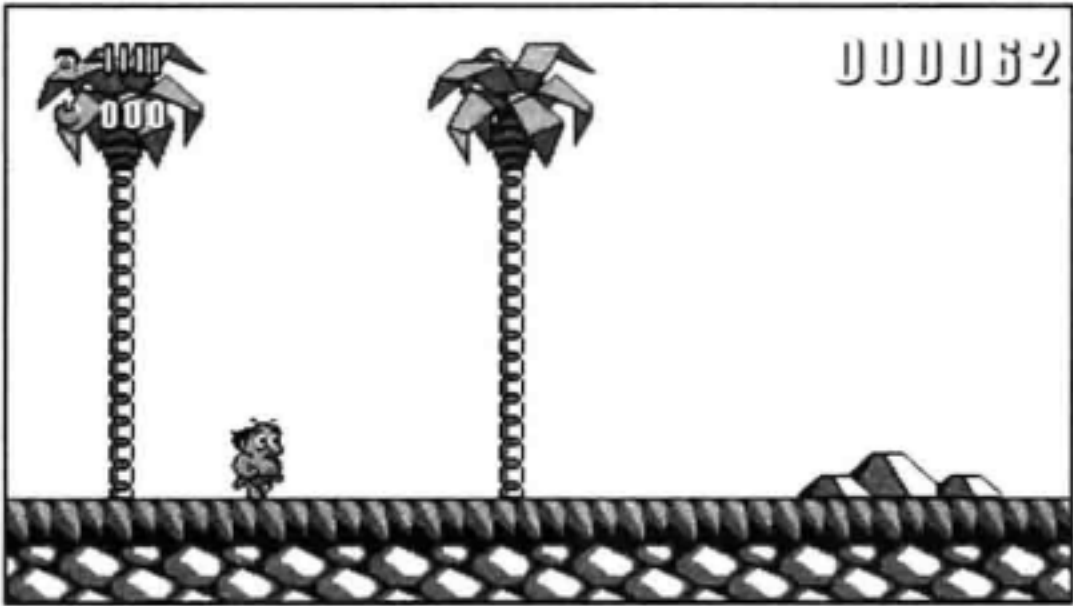


图 5.47 第一关的场景

当玩家选择第二关时，就会由选择关卡的场景跳转到到第二关的场景中，如图 5.48 所示。

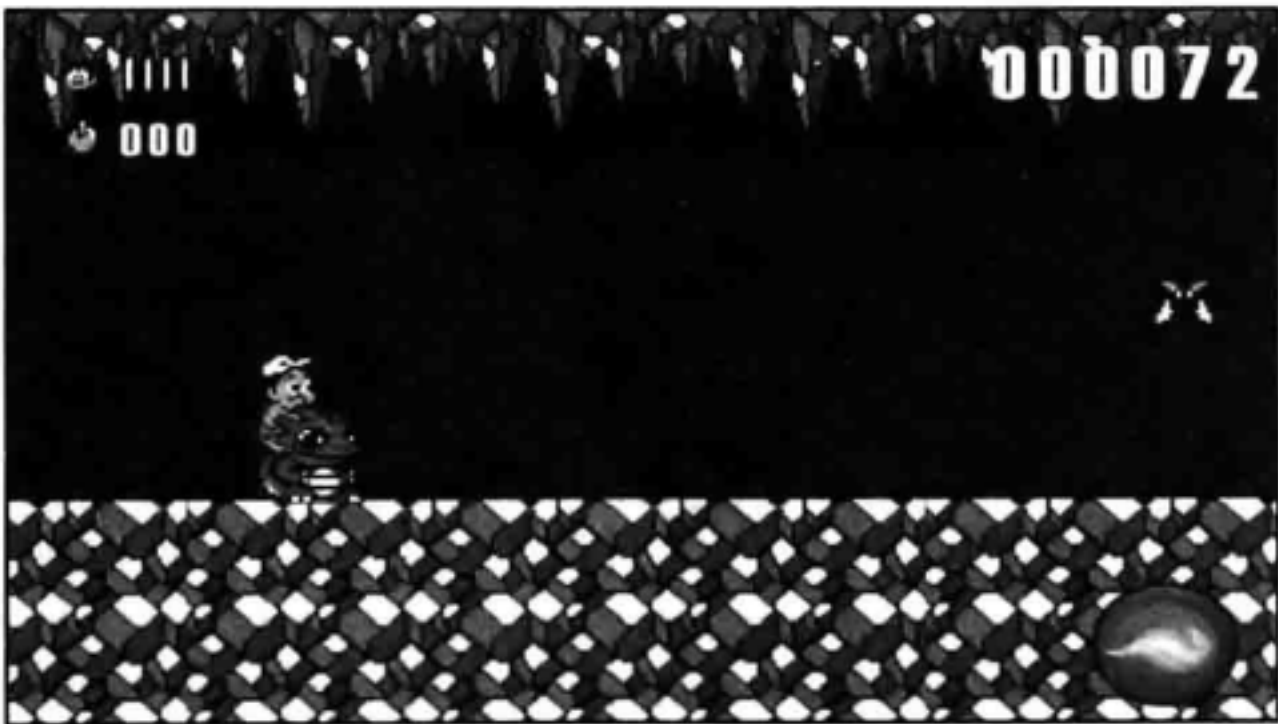


图 5.48 第二关的场景

此时，它们就使用到了场景与场景的切换功能，即当玩家轻拍相应的场景后，就会进入到所选场景中。

5.9.2 实现场景切换

以轻拍 Play Game 按钮进入到 Game Scene View Controller 视图控制器的场景为例，它切换步骤如下所述。

(1) 单击实现主菜单的视图控制器，在此视图控制器中，选择界面上方的 Dock 中的 View Controller 图标，在工具窗口中的 Show the Attributes inspector 选项，即属性查看器选中，找到 Is Initial View Controller 复选框，将其选中，使此视图控制器称为初始视图控制器。

(2) 按住 Ctrl 键拖动 View Controller 视图控制器的界面中的 Play Game 按钮对象，这时会出现一个蓝色的线条，将这个蓝色的线条和 Game Scene View Controller 视图控制器的界面进行关联，如图 5.49 所示。

(3) 松开鼠标后，会弹出一个 Action Segue 对话框，如图 5.50 所示。

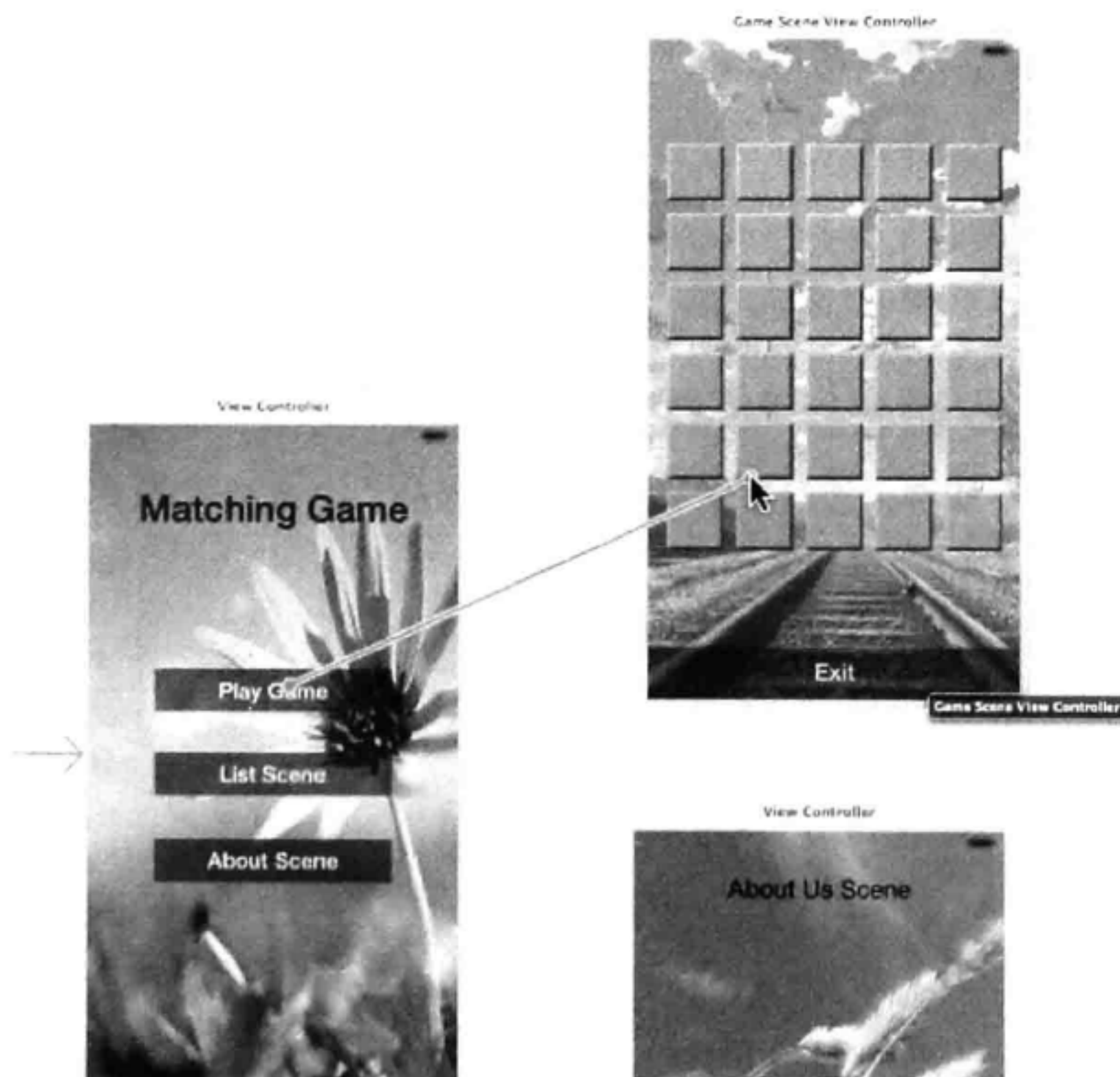


图 5.49 操作步骤 1



图 5.50 操作步骤 2

注意：在图 5.50 中出现的 Action Segue 窗口中有 3 个类型，分别为 push、modal、custom，以下是这 3 个类型的介绍。

- ☐ push: 一般是需要第一个界面是 Navigation Controller 导航控制器。
- ☐ modal: 模态转换，一般用在视图的切换中。
- ☐ custom: 用于自定义跳转方式。

(4) 选择其中的 modal 选项，这时新增一个箭头，如图 5.51 所示。

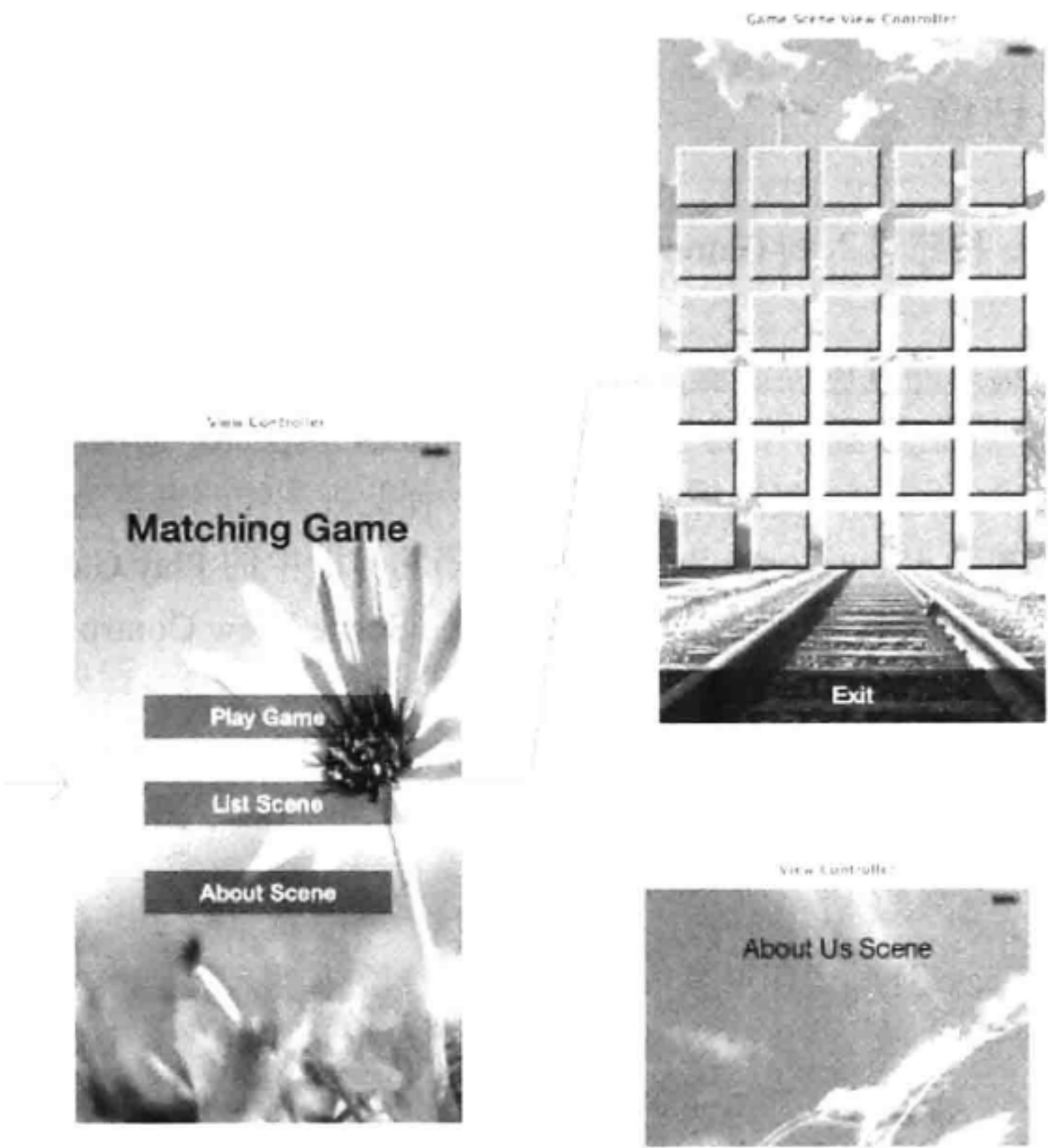


图 5.51 操作步骤 3

🔔注意：这时在图中会出现两个箭头。左边的箭头是一直存在的，它是开始箭头，右边的箭头就是 Segue 箭头，这个箭头表示视图的切换。此时，运行程序，当轻拍 Play Game 按钮后，可以看到场景的切换，如图 5.52 所示。

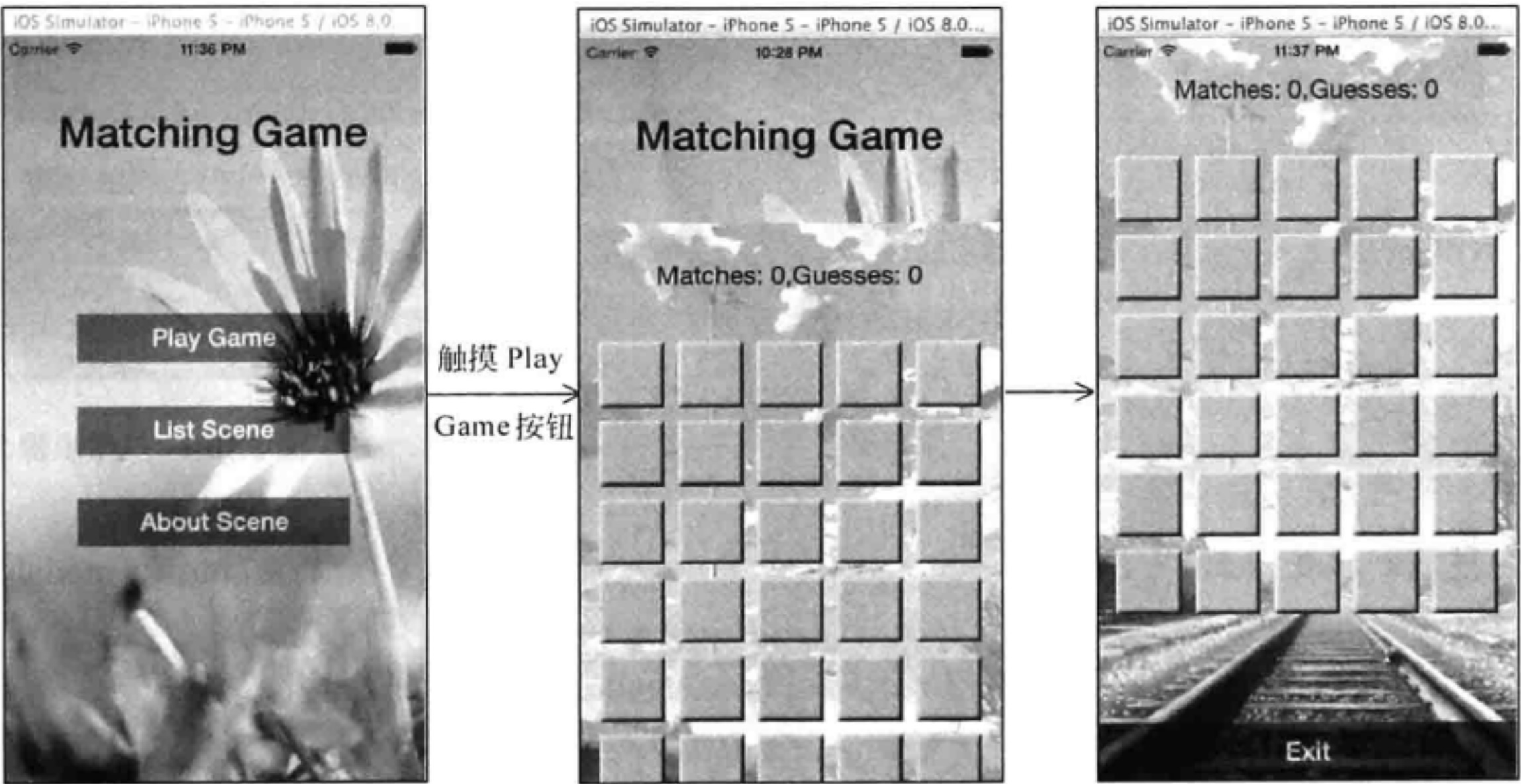


图 5.52 运行效果

5.9.3 过渡动画效果

在场景与场景的切换期间，可以看到场景不是直接进行切换的，而是使用了过渡动画效果。这个过渡动画效果是可以进行设置的。选择产生的 Segue 箭头，使箭头成蓝色，蓝色表示此箭头正在编辑。选择工具窗口的 Show the Attributes inspector 选项，即属性查看器，在其中找到 Transition 属性。此属性就是对过渡动画效果进行设置的，如图 5.53 所示。

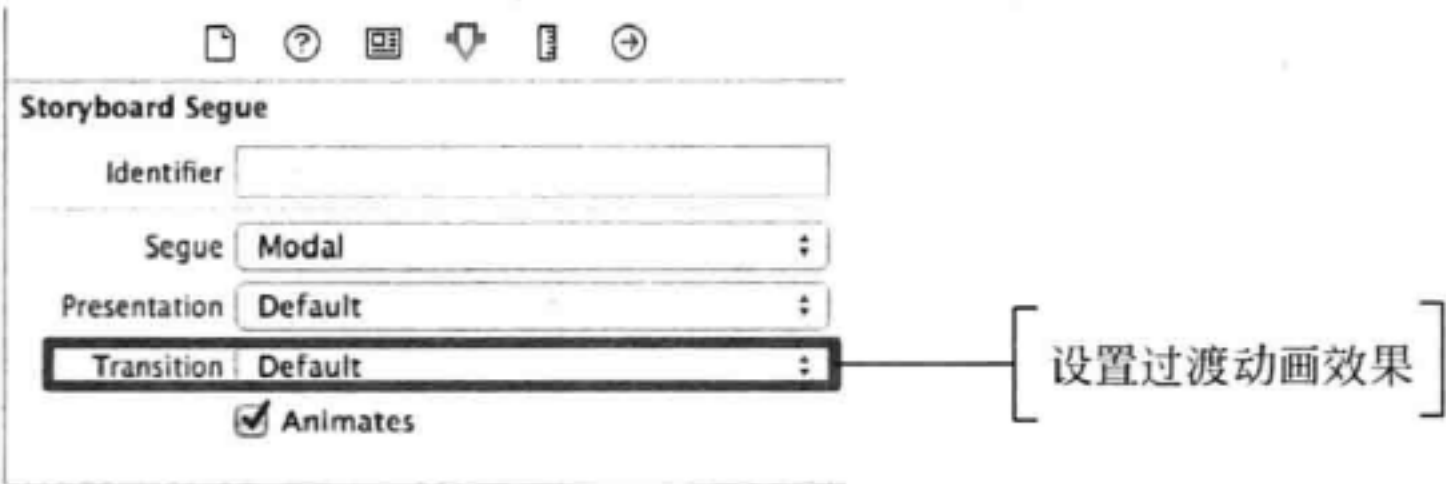


图 5.53 设置过渡动画效果

Transition 属性可以设置的过渡动画包括 5 种。

1. Default或者Cover Vertical

Default 动画效果就是 Cover Vertical 动画效果，它们可以实现垂直覆盖的动画效果，如图 5.54 所示。

2. Flip Horizontal

Flip Horizontal 可以实现水平翻转的动画效果，如图 5.55 所示。

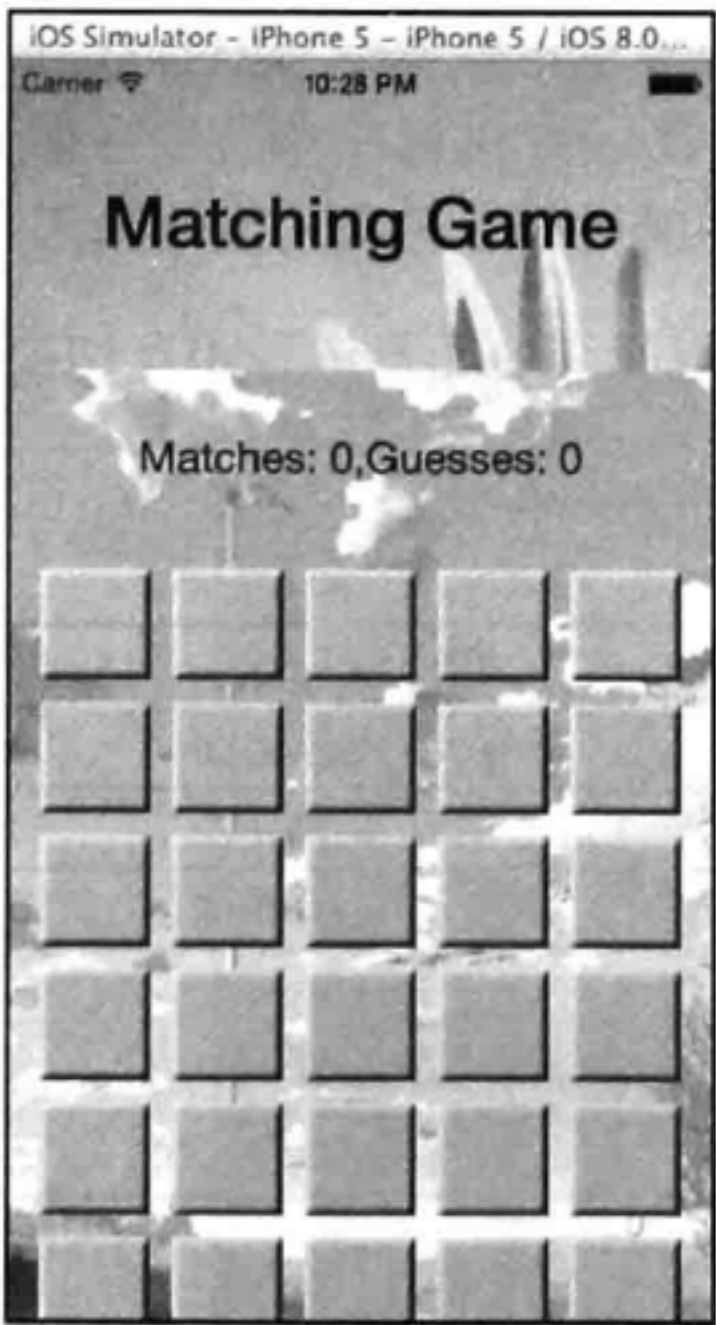


图 5.54 Default 或者 Cover Vertical 动画效果

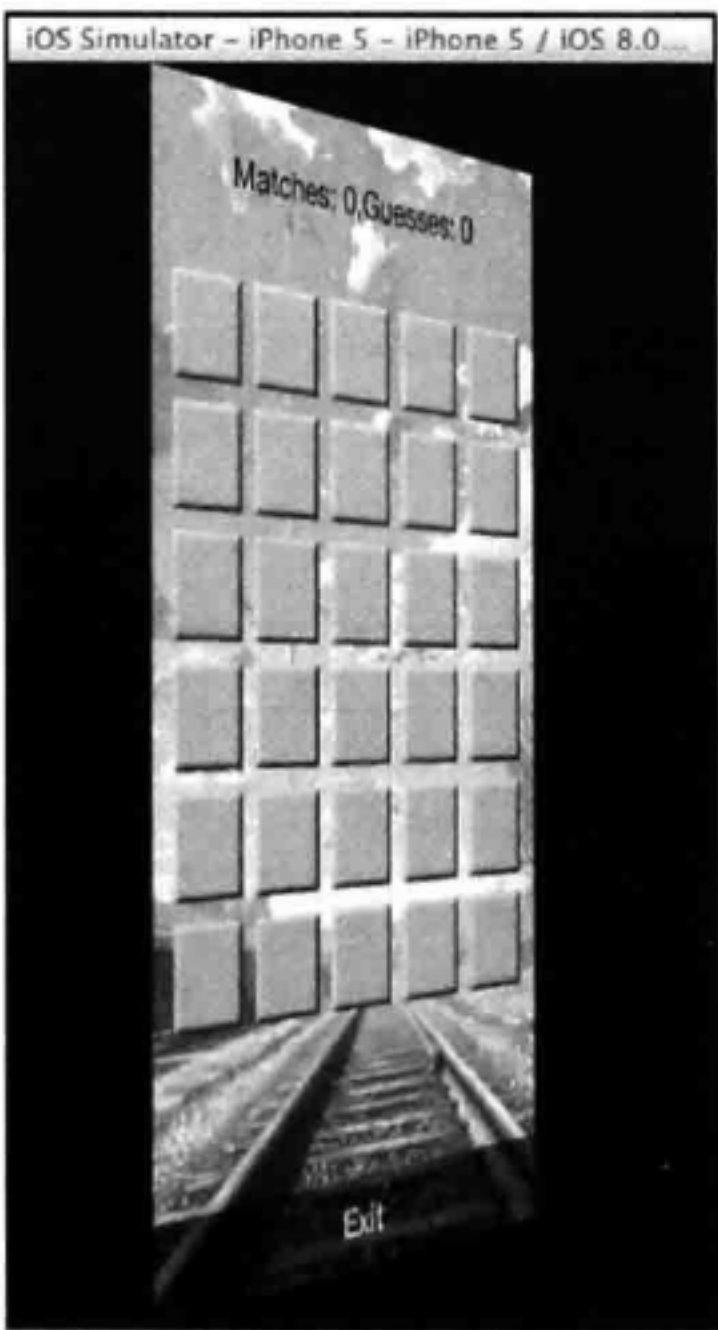


图 5.55 Flip Horizontal 动画效果

3. Cross Dissolve

Cross Dissolve 可以实现淡入和淡出的动画效果，如图 5.56 所示。

4. Partial Curl

Partial Curl 可以实现翻页的动画效果，如图 5.57 所示。

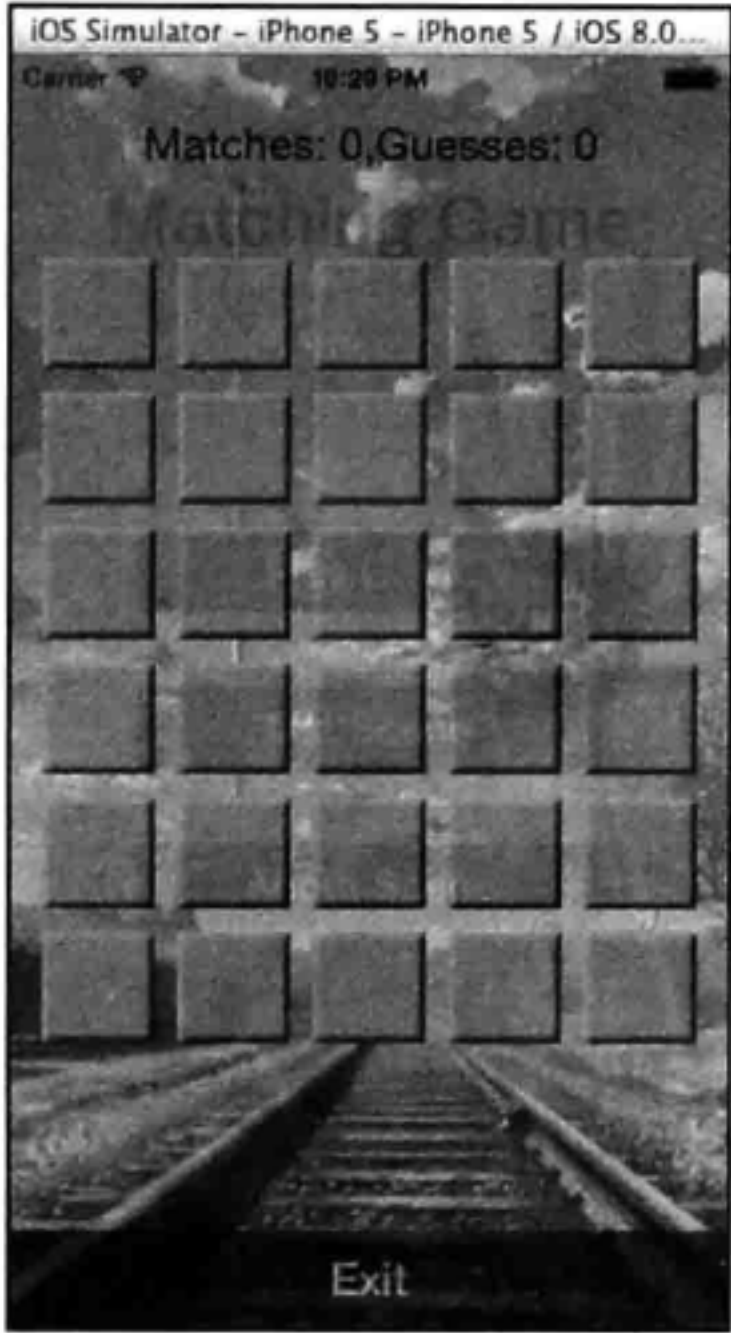


图 5.56 Cross Dissolve 动画效果

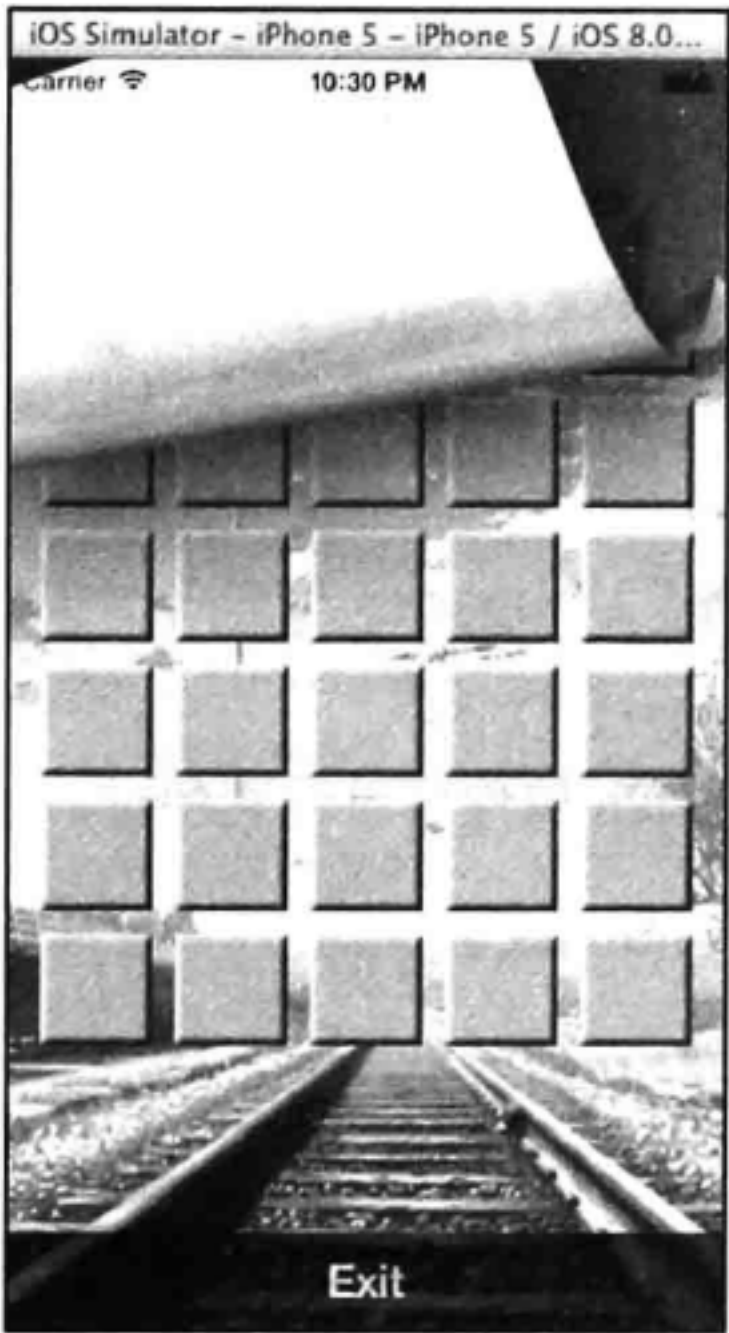


图 5.57 Partial Curl 动画效果

5.9.4 全部的场景切换

以上只是针对一个场景的切换，在此游戏中需要将所有的场景进行相互切换。具体的切换关系如表 5-6 所示。

表 5-6 场景切换

对 象	切 换 到
Play Game 按钮	配对游戏的界面
List Scene 按钮	分数榜单的界面
About Scene 按钮	关于游戏的界面
配对游戏界面中的 Exit 按钮	主菜单的界面
分数榜单的界面中的 Exit 按钮	
关于游戏的界面中的 Exit 按钮	

⚠注意：它们的切换步骤都是一样的。

最后，画布中的效果如图 5.58 所示。

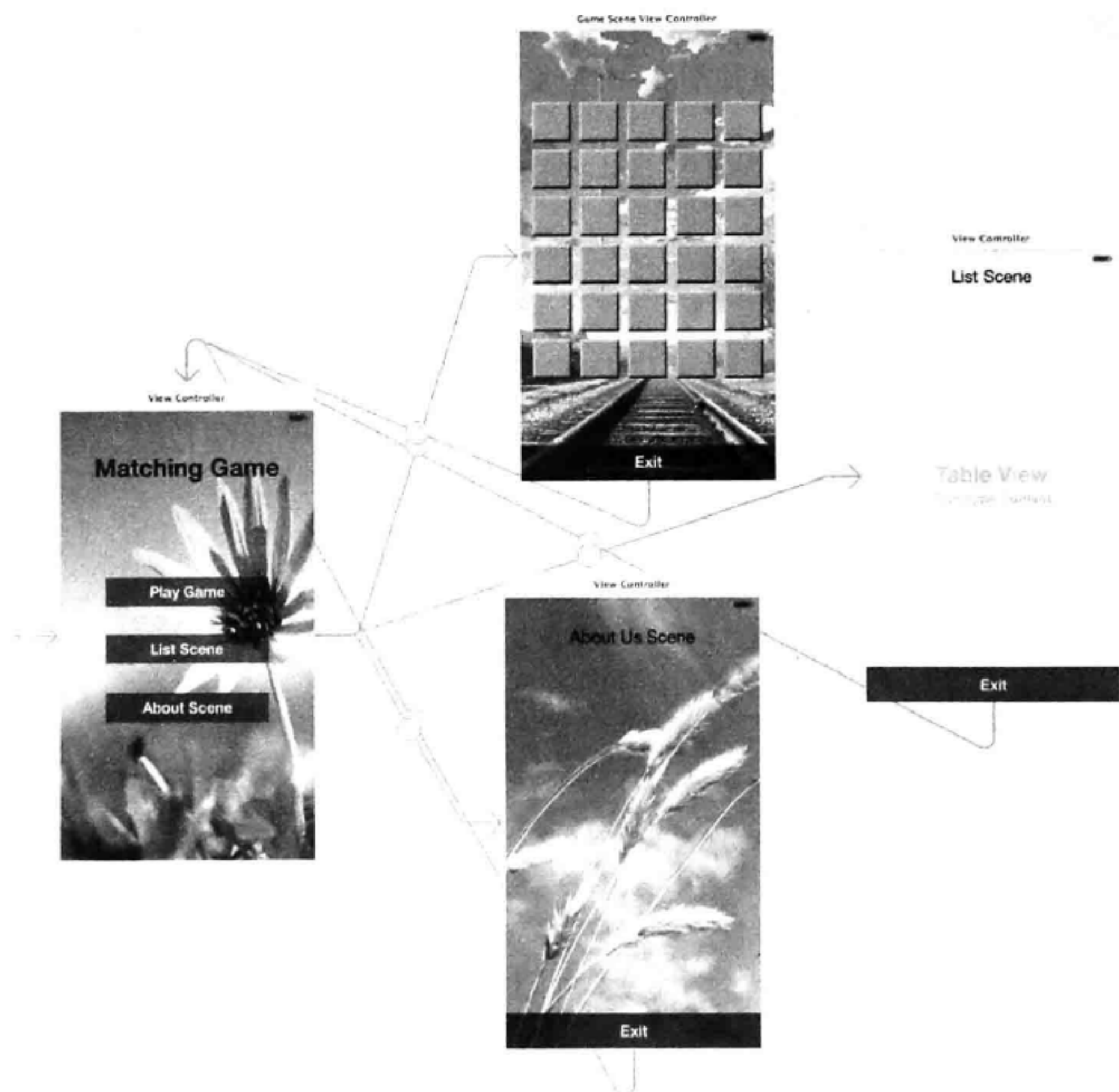


图 5.58 画布效果

此时运行程序，会看到以下的效果。

当玩家在主菜单中轻拍 Play Game 按钮，可以进入配对游戏的界面。当玩家轻拍 Exit 按钮，可以返回到主菜单，效果如图 5.59 所示。



图 5.59 运行效果 1

当玩家在主菜单中轻拍 List Scene 按钮，可以进入分数榜单的界面。当玩家轻拍 Exit 按钮，可以返回到主菜单，效果如图 5.60 所示。

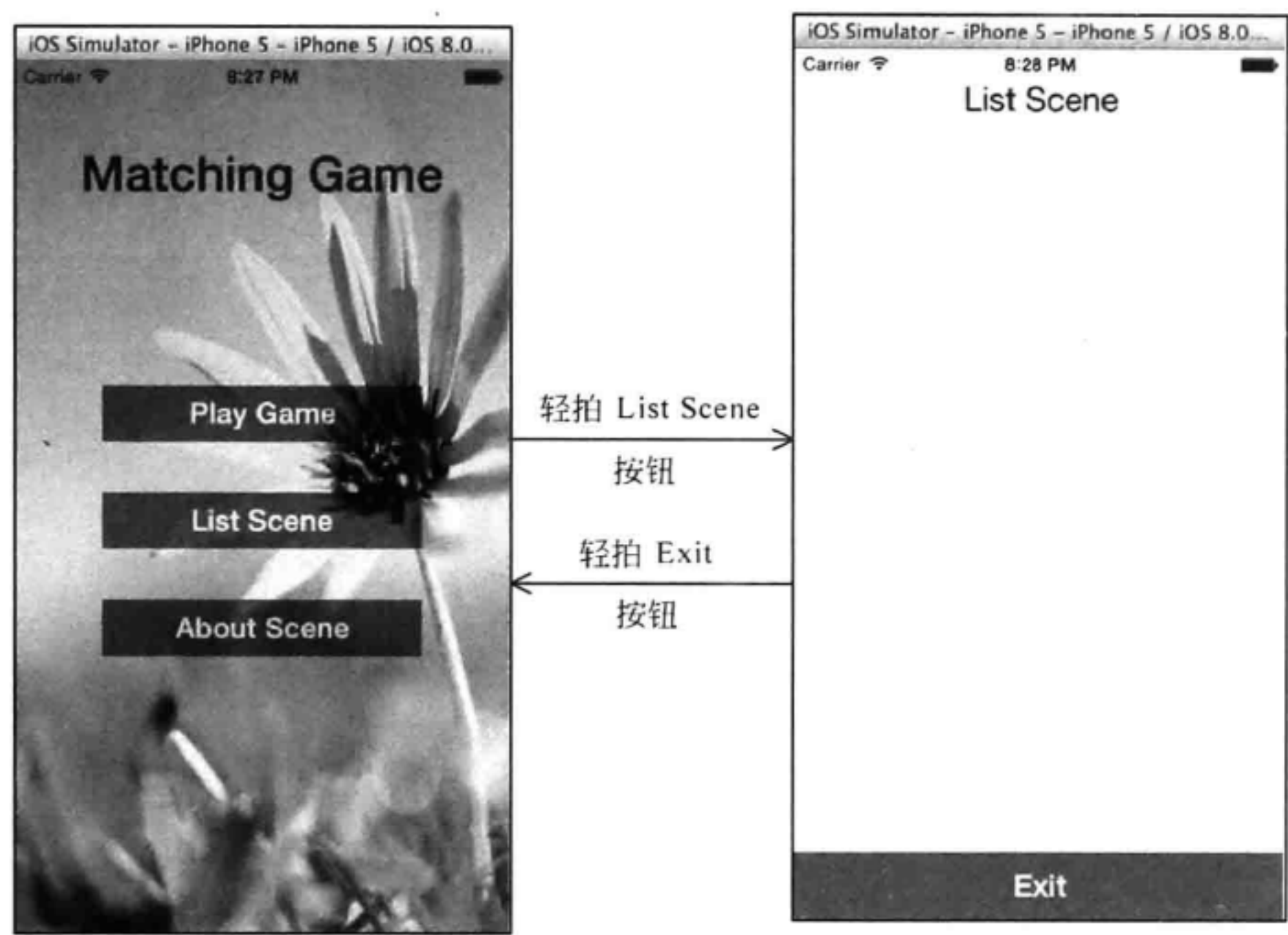


图 5.60 运行效果 2

当玩家轻拍 About Scene 按钮，可以进入关于游戏的界面。当玩家轻拍 Exit 按钮，可以返回到主菜单，效果如图 5.61 所示。

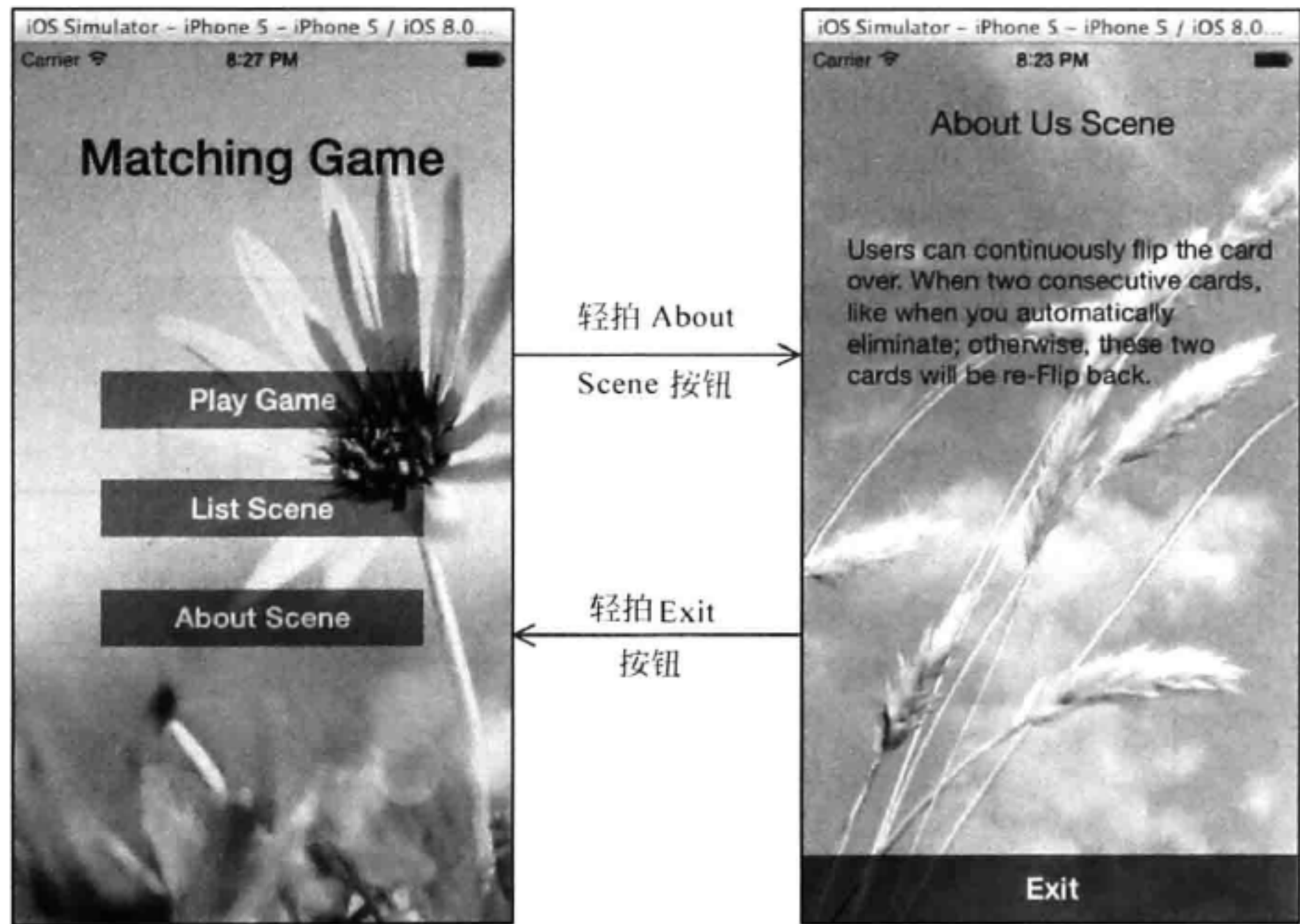


图 5.61 运行效果

第 6 章 太空侵略者——绘制图像

太空侵略者的名字大家比较陌生。在国内，它通常被称为小蜜蜂。小蜜蜂作为一款街机游戏，曾经在 30 年前风靡国内。本章也将制作一款太空侵略者游戏。通过本章内容，将讲解图形绘制和图形移动等技术。

6.1 游戏介绍

太空侵略者是一款历史悠久的经典射击游戏系列。玩家操作以 2D 点阵图构成的太空船，在充满外星侵略者的太空中进行一连串的抵抗任务。玩家除了能左右移动太空飞船来闪躲敌人，还可以发射子弹对侵略者进行射击。这样的游戏，通常包括以下几个模块。

1. 主菜单模块

和所有的主菜单一样，太空侵略者游戏的主菜单也提供一个基本的界面，帮助用户操作，如图 6.1 所示。轻拍 Play Game 按钮，可以进入射击游戏的界面。

2. 射击游戏模板

射击游戏模块是本程序最重要的模块。它提供了用户游戏的界面，如图 6.2 所示。同时，它会负责对用户的操作做出响应。

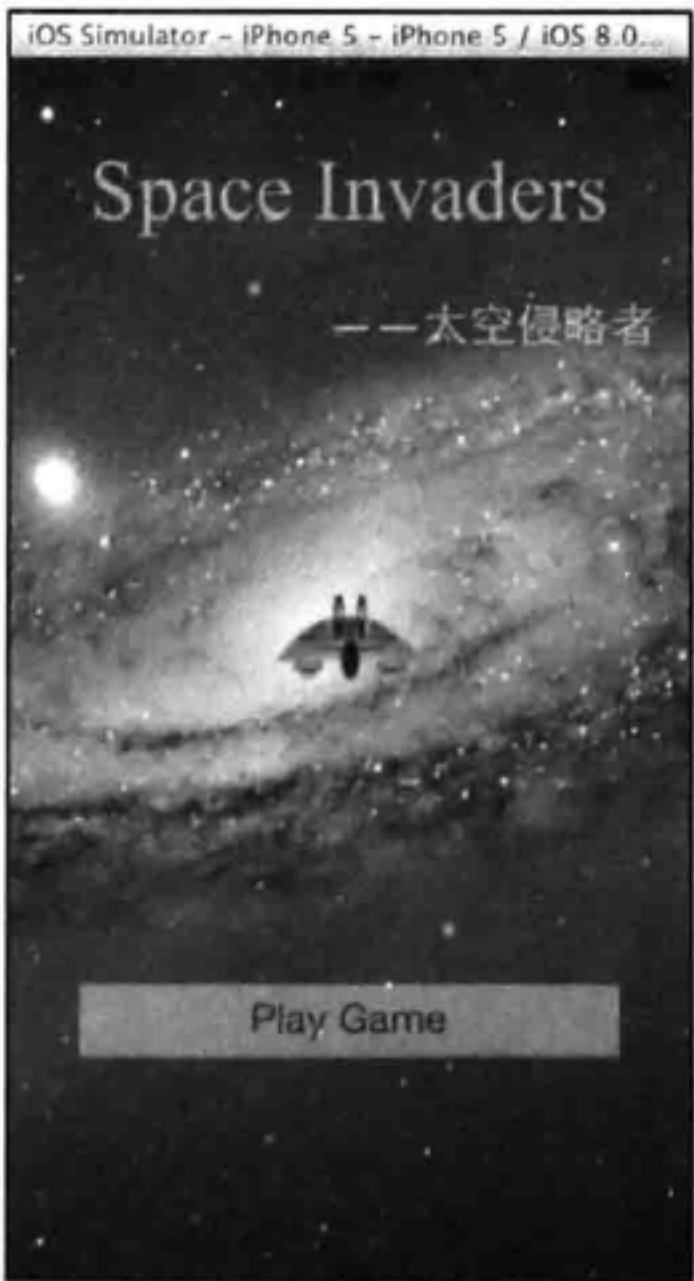


图 6.1 主菜单界面

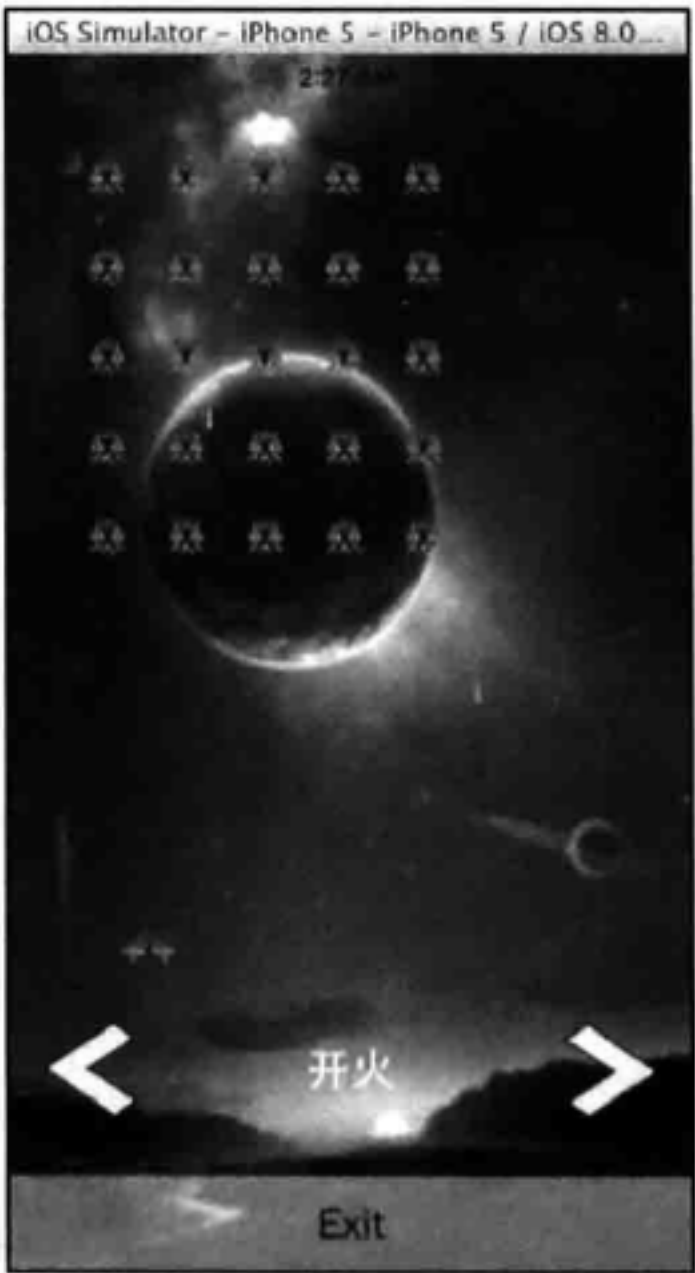


图 6.2 游戏界面

6.2 开发游戏之前的准备工作

在开发太空侵略者游戏之前，需要做一些准备工作，这些准备工作如下所述。

1. 创建工程

创建一个 Single View Application 模板类型的项目，命名为 Space SpaceInvaders。

2. 添加图像

添加图像 backdrop.jpg、background.jpg、left.png、right.png、bullet.png、enemy01.png 和 ship.png 到创建项目的 Supporting Files 文件夹中，如图 6.3 所示。

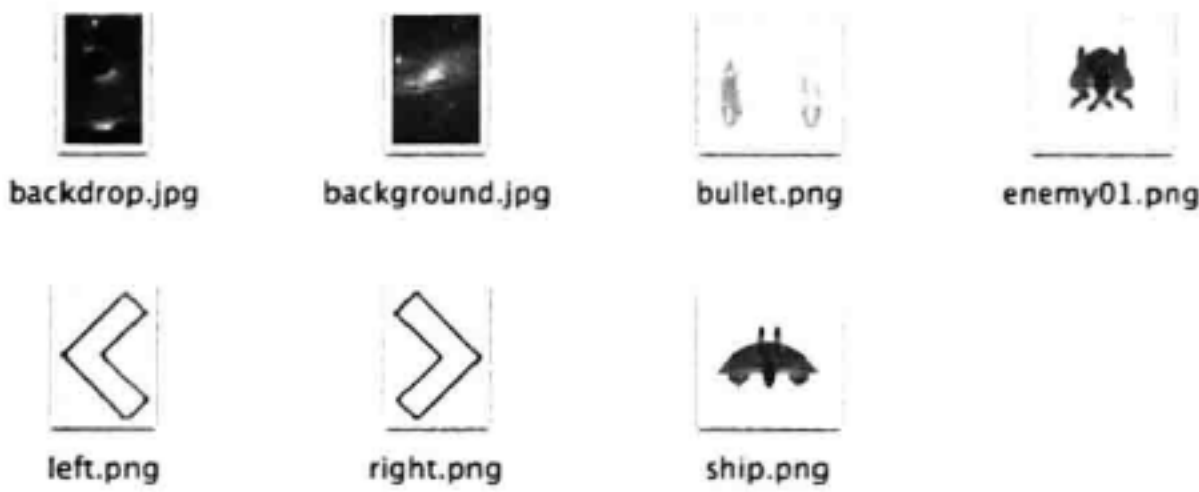


图 6.3 添加的图像

6.3 主菜单模板

在每一个游戏应用程序中都有一个主菜单，当然，本章的游戏也不例外。对于主菜单的菜单项，还是需要使用按钮来实现。本节将讲解主菜单的设计。

双击鼠标将 Main.storyboard 文件打开，对主菜单的界面（界面也可以称为主视图或者场景）进行设计，具体的操作步骤如下所述。

（1）选择 Show the File inspector 选项，将 Interface Builder Document 中的 Opens in 改为 Xcode 5.1。

（2）对在画板中原本存在的 View Controller 视图控制器的界面进行设计，效果如图 6.4 所示。

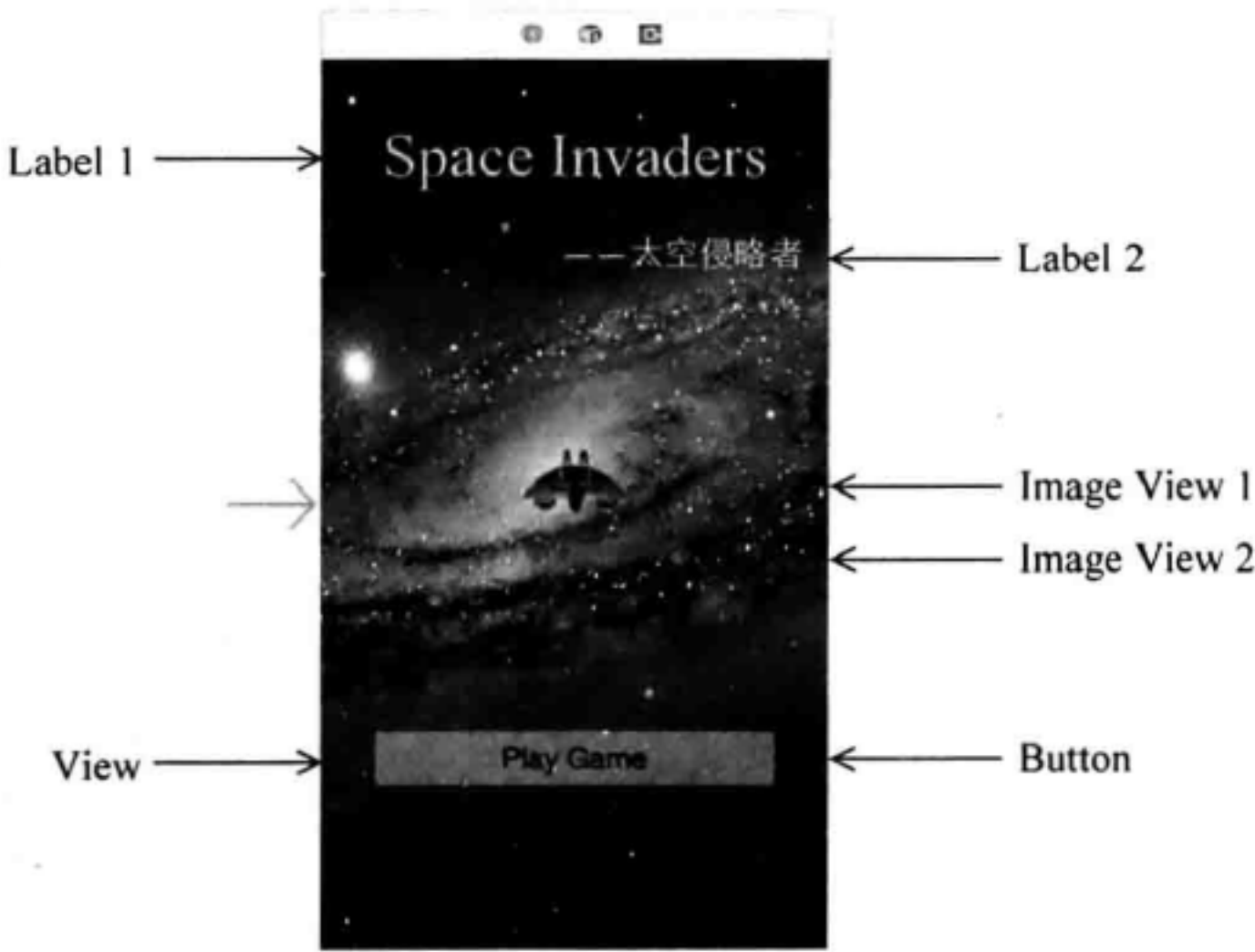


图 6.4 界面的效果

需要添加的视图对象，以及对它们的设置，如表 6-1 所示。

表 6-1 设置界面	
视 图	设 置
Label1	Text: Space Invaders Color: 绿色 Font: Times New Roman 40.0 Alignment: 居中 位置和大小: (22, 28, 276, 67)
Label2	Text: ——太空侵略者 Font: System 22.0 Alignment: 居右 位置和大小: (113, 114, 191, 21)
Image View1	Image: background.jpg 位置和大小: (0, 0, 320, 568)
Image View2	Image: ship.png 位置和大小: (119, 229, 83, 82)
View	Alpha: 0.4 位置和大小: (34, 431, 252, 34)
Button	Title: Play Game Font: System 19.0 Text Color: 黑色 位置和大小: (35, 2, 182, 29)

6.4 射击游戏模板

射击游戏模块是本程序的最重要的模块。它提供了用户游戏的界面。本节将讲解射击游戏的准备工作和对射击游戏界面的设计。

6.4.1 准备工作

在设计射击游戏界面前，需要做一些准备工作，例如，为了便于代码的管理，将每一个界面实现的功能代码保存在一个文件中。

1. 创建文件

在创建的项目中需要创建一个 Swift File 模板类型的文件，命名为 Game.swift。

2. 创建空类

打开创建的 Game.swift 文件，在此文件中创建一个基于 UIViewController 类的空类 GameViewController，代码如下：

```
import UIKit
class GameViewController: UIViewController {
```

```
.....
}
```

该类用于编写实现射击游戏的代码。

6.4.2 设计界面

以下是对游戏界面进行设计的具体操作步骤。

(1) 在视图对象库中拖动 View Controller 视图控制器对象到画布中。

(2) 单击新添加的视图控制器，选择界面上方的 Dock 中的 View Controller 图标。在工具窗口中的 Show the Identity inspector 选项，即属性查看器中，将 Custom Class 下的 Class 设置为创建的 GameViewController 类。这时，在画布中的这个视图控制器就变为了 Game View Controller 视图控制器。

(3) 对 Game View Controller 视图控制器的界面进行设计，效果如图 6.5 所示。

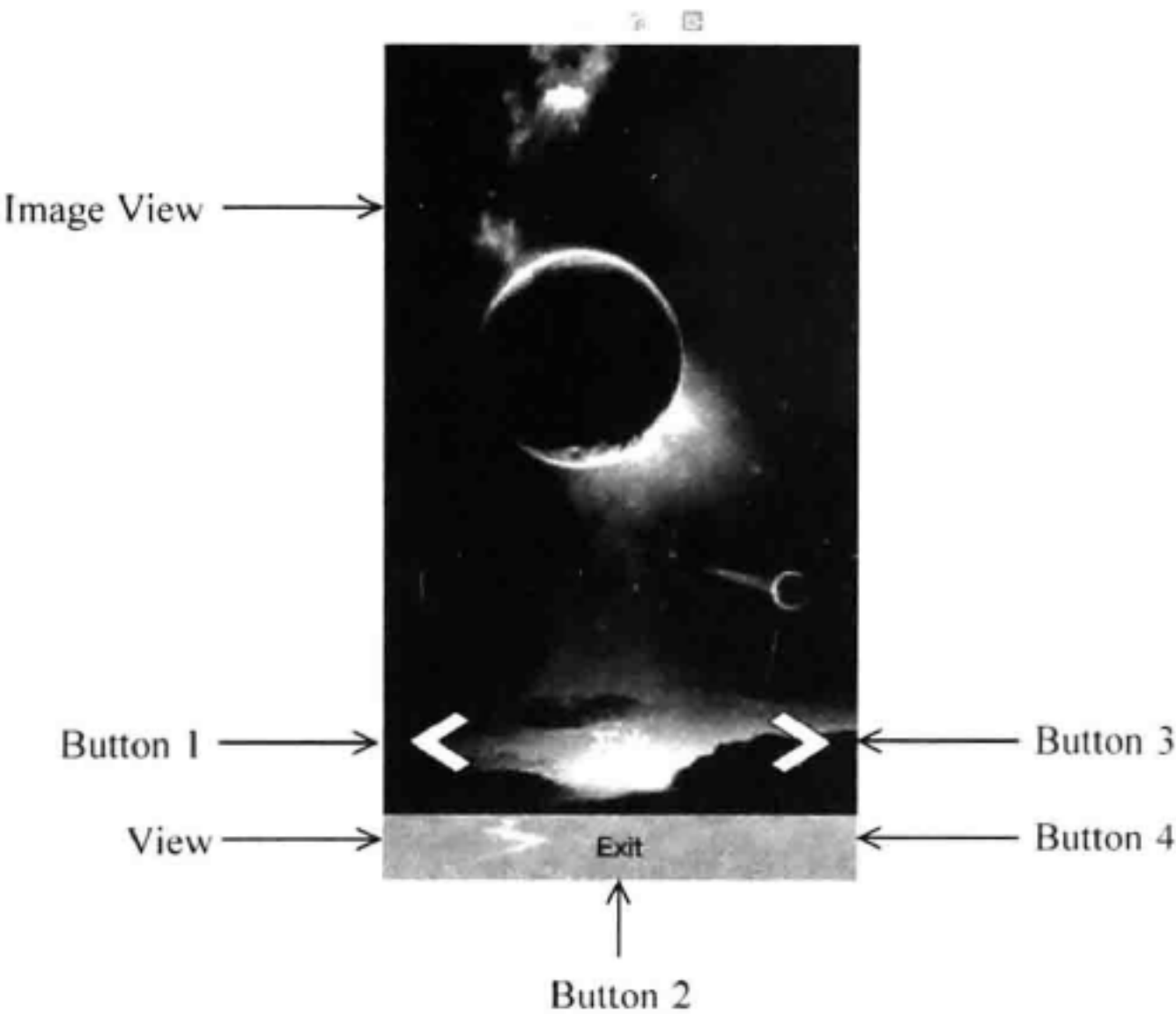


图 6.5 界面的效果

需要添加的视图对象，以及对它们的设置，如表 6-2 所示。

表 6-2 设置界面

视 图	设 置
Image View	Image: background.jpg 位置和大小: (0, 0, 320, 568)
Button1	Title: (空) Background: left.png 位置和大小: (16, 452, 45, 44)
Button2	Title: 开火 Font: System Bold 21.0 Text Color: 白色 位置和大小: (137, 459, 46, 30)

续表

视 图	设 置
Button3	Title: (空) Background: right.png 位置和大小: (259, 452, 45, 44)
View	Alpha: 0.5 位置和大小: (0, 523, 320, 45)
Button4	Title: Exit Font: Helvetica Neue 19.0 Text Color: 黑色 位置和大小: (137, 8, 46, 30)

6.5 添加飞船

在太空侵略者的游戏中，一般使用飞船来表示正义的一面，即玩家可以控制的一方。本节将使用代码实现飞船的添加。

飞船实际上是一个图像，如果想要在界面显示这个图像，需要添加一个图像视图，将图像视图的 `Image` 属性设置为飞船的图像就可以了。在前面的章节中，为一个界面添加视图对象都是以拖动视图对象库中的视图对象进行的，而不曾使用过代码。使用代码添加视图对象的方式一般称为动态创建视图。为一个界面动态地创建视图需要通过以下几个步骤。

1. 实例化视图对象

简单地说，视图是使用类实现的。对于一个类的使用，往往需要进行实例化，其语法形式如下：


```
var 视图对象名:视图类=视图类()
```

2. 设置视图框架

每一个视图都是有位置和大小，所以在实例化视图对象后，需要对此视图的框架进行设置，其框架包含了位置和大小。框架的设置一般需要使用到 `frame` 属性。其语法形式如下：

```
视图对象名.frame= CGRectMake(x:CGFloat, y:CGFloat, width:CGFloat, height:CGFloat)
```

其中，`x` 和 `y` 用来指定视图的所在 `X` 轴和 `Y` 轴的位置，`width` 和 `height` 用来指定视图的宽和高。

 **注意：**一般可以将实例化对象和设置视图框架放在一起进行编写，其语法形式如下所示。

```
var 视图对象名:视图类=视图类(frame: CGRectMake(x:CGFloat, y:CGFloat, width:CGFloat, height:CGFloat))
```

3. 添加视图

最后，需要将视图对象添加到界面中，即主视图中，需要使用到 `addSubview()` 方法。其语法形式如下：

```
self.view.addSubview(视图对象名)
```

打开 `Game.swift` 文件，添加 `viewDidLoad()` 方法，在此方法中实现飞船的添加。代码如下：

```
import UIKit
class GameViewController: UIViewController {
    var playerImage:UIImage=UIImage(named: "ship.png")
    var playerRect:CGRect=CGRectMake(0, 0, 0, 0)
    var playerView:UIImageView=UIImageView()
    override func viewDidLoad() {
        super.viewDidLoad()
        self.playerView.image=self.playerImage           //设置显示的图像
        self.playerRect=CGRectMake(50, 400, 32, 32)
        self.playerView.frame=self.playerRect           //设置框架
        self.view.addSubview(self.playerView)           //添加视图
    }
}
```

回到 `Main.storyboard` 文件中。单击 `Game View Controller` 视图控制器。在此视图控制器中，选择界面上方的 Dock 中的 `Game View Controller` 图标。在工具窗口中的 `Show the Attributes inspector` 选项，即属性查看器中，找到 `Is Initial View Controller` 复选框，将其选中。此时 `Game View Controller` 视图控制器就成为了初始视图控制器。此时运行程序，可以看到如图 6.6 所示的效果。

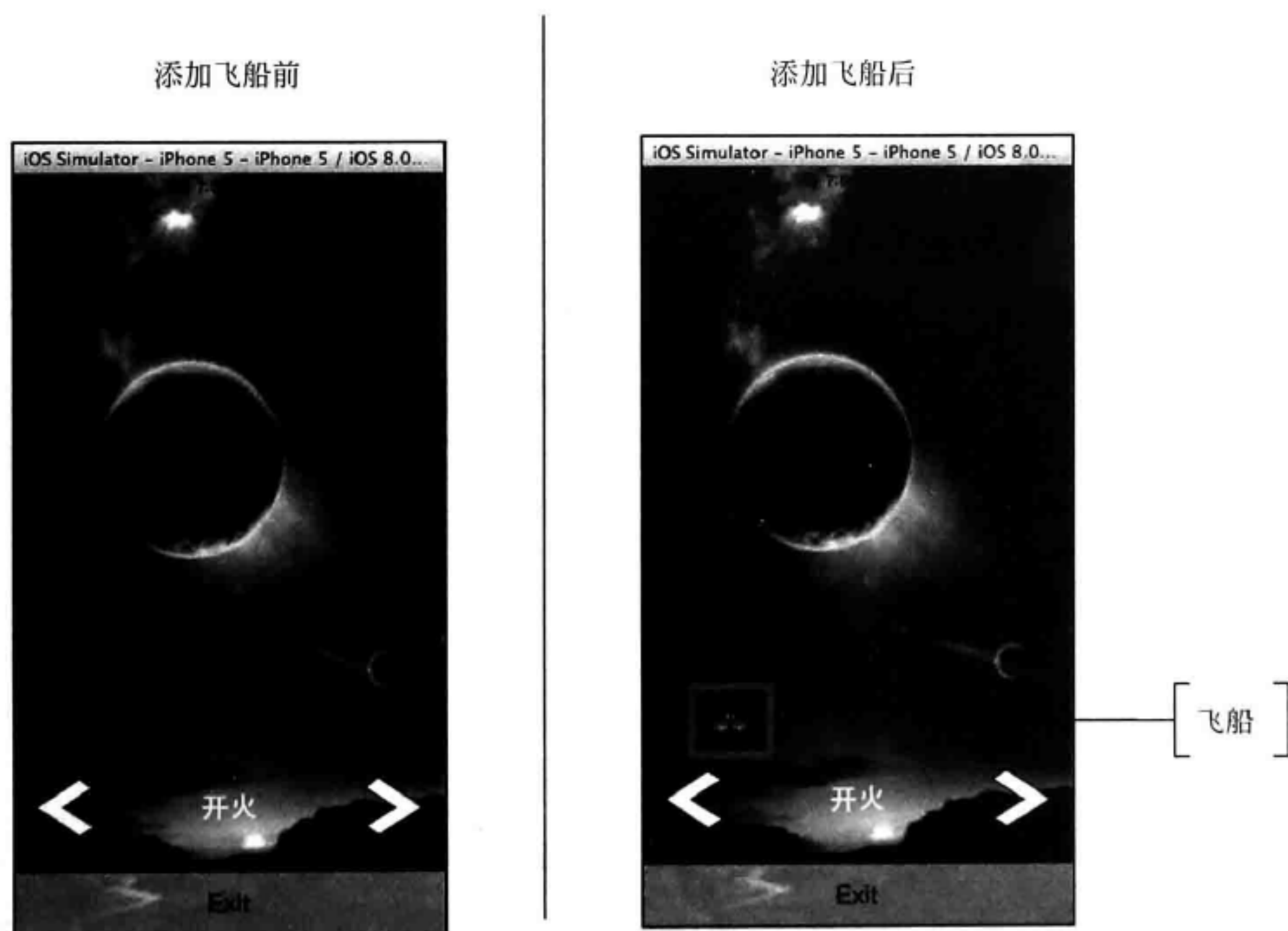


图 6.6 运行效果

6.6 移动飞船

在常见的太空侵略者游戏中，飞船并不是固定的。它是可以移动的，并且由玩家来控制飞船移动的方向。本节将讲解飞船是如何实现移动的。

6.6.1 向左移动

在射击游戏界面，我们添加了3个按钮。其中，带有箭头的按钮是用来控制飞船向左移动或向右移动的。本小节将讲解如何实现飞船的向左移动。其实它的实现思路很简单，就是当玩家轻拍向左的按钮后，开始执行向左移动动画。在编写飞船向左移动的代码前，首先需要为按钮关联一个 `moveLeft()` 的动作，当玩家轻拍向左的按钮后，就可以触发此动作（动作的关联在上一章中讲解过了）。关联好动作后，就可以实现飞船的移动动画了，此时的动画需要使用定时器来实现，所以需要在 `moveLeft()` 动作中添加以下的代码：

```
@IBAction func moveLeft(sender: AnyObject) {
    moveTimer=NSTimer.scheduledTimerWithTimeInterval(0.03, target: self,
    selector: Selector("movePlayerLeft"), userInfo: nil, repeats: true)
}
```

在创建的定时器中，可以看到，每隔 0.03 秒，都会自动调用一个叫 `movePlayerLeft()` 的方法，此方法实现的就是向左移动的功能。所以需要在 `moveLeft()` 动作的后面添加 `movePlayerLeft()` 的方法。其添加的代码如下：

```
func movePlayerLeft() {
    //判断飞船是否超出边界
    if(self.playerRect.origin.x >= 10){
        self.playerRect = CGRectOffset(self.playerRect, -3, 0);
        self.playerView.frame = self.playerRect;
    }
}
```

此时运行程序，可以看到如图 6.7 所示的效果。

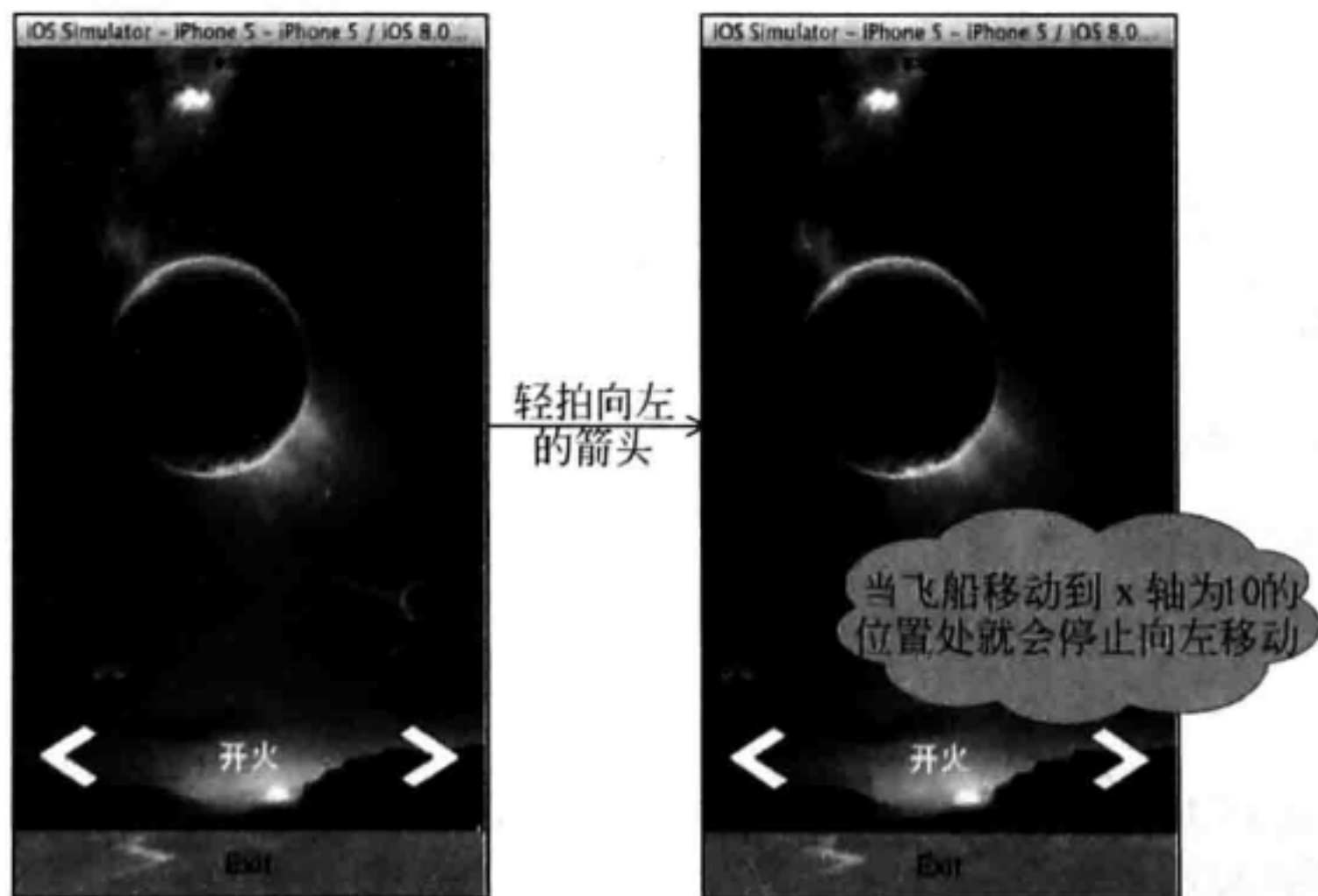


图 6.7 运行效果

6.6.2 向右移动

飞船的向右移动和向左移动是一样的。首先需要为按钮关联一个 `moveRight()` 的动作，当玩家轻拍向右的按钮后，就可以触发此动作。关联好动作后，就可以实现飞船的移动动画了，此时的动画也需要使用定时器实现，所以需要在 `moveRight()` 动作中添加以下的代码：

```
@IBAction func moveRight(sender: AnyObject) {
    moveTimer=NSTimer.scheduledTimerWithTimeInterval(0.03, target: self,
    selector: Selector("movePlayerRight"), userInfo: nil, repeats: true)
}
```

在创建的定时器中可以看到，每隔 0.03 秒，会自动调用一个叫 `movePlayerRight()` 的方法，此方法实现的就是向右移动的功能。所以需要在个 `moveRight()` 动作的后面添加 `movePlayerRight()` 的方法，其添加的代码如下：

```
func movePlayerRight() {
    //判断飞船是否超出了边界
    if(self.playerRect.origin.x <= 290){
        self.playerRect = CGRectOffset(self.playerRect, 3, 0);
        self.playerView.frame = self.playerRect;
    }
}
```

此时运行程序，可以看到如图 6.8 所示的效果。

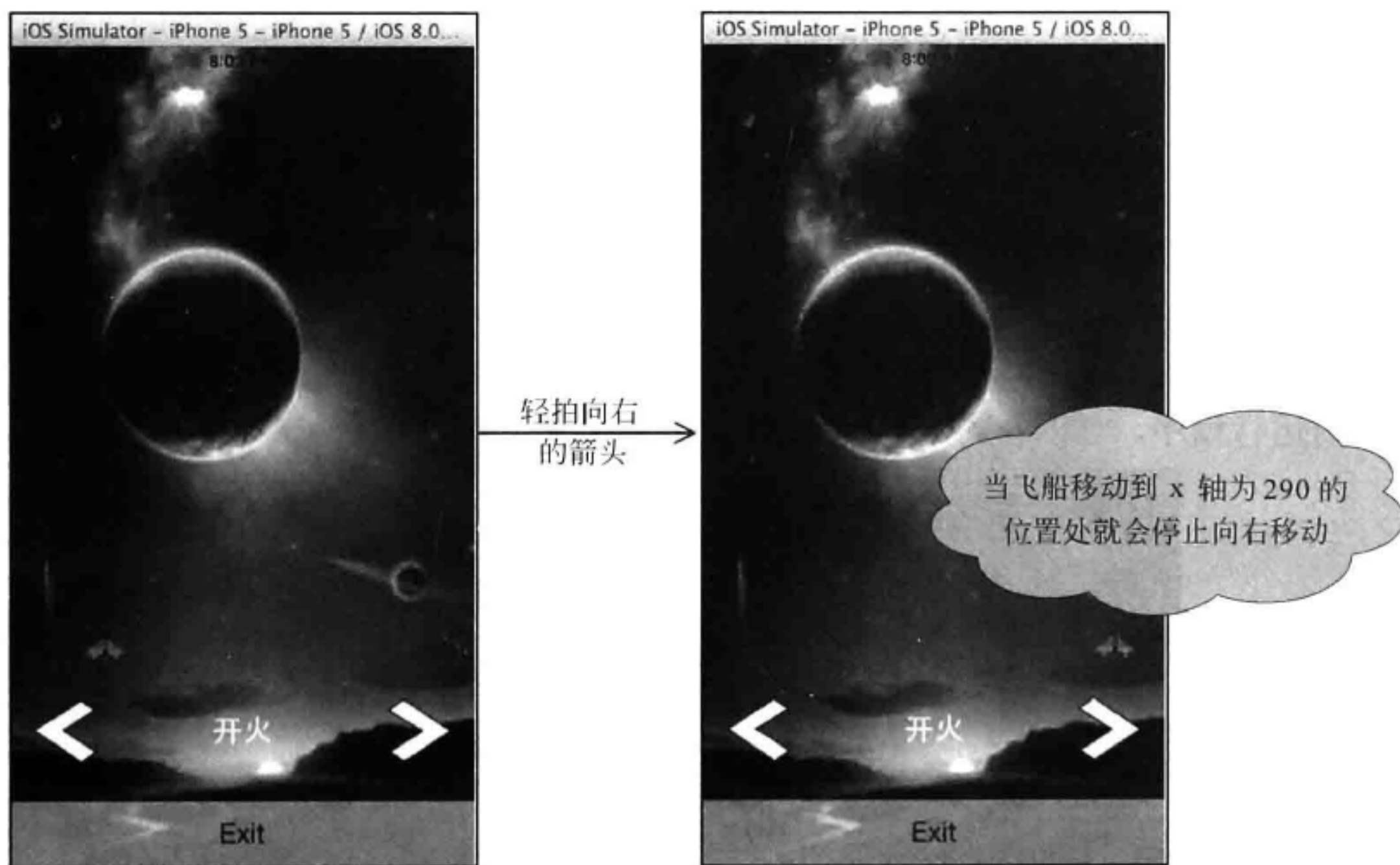


图 6.8 运行效果

注意：当再一次轻拍向左的按钮后，由于向右的动画还在继续，此时飞船就会处于混乱状态，不会向左移动。此时，需要将向右的动画终止，这时就需要添加一个终止动画的方法，代码如下：

```
func releaseTouch() {
    //判断定时器是否为空
    if(self.moveTimer != nil){
        self.moveTimer?.invalidate()           //终止定时器
        self.moveTimer = nil;
    }
}
```

此方法需要在 moveLeft 和 moveRight 动作中进行调用,使它可以在每轻拍一次按钮后,执行一次新的动画。调用的代码如下:

```
Self.releaseTouch()
```

此时再一次运行程序,可以看到如图 6.9 所示的效果。

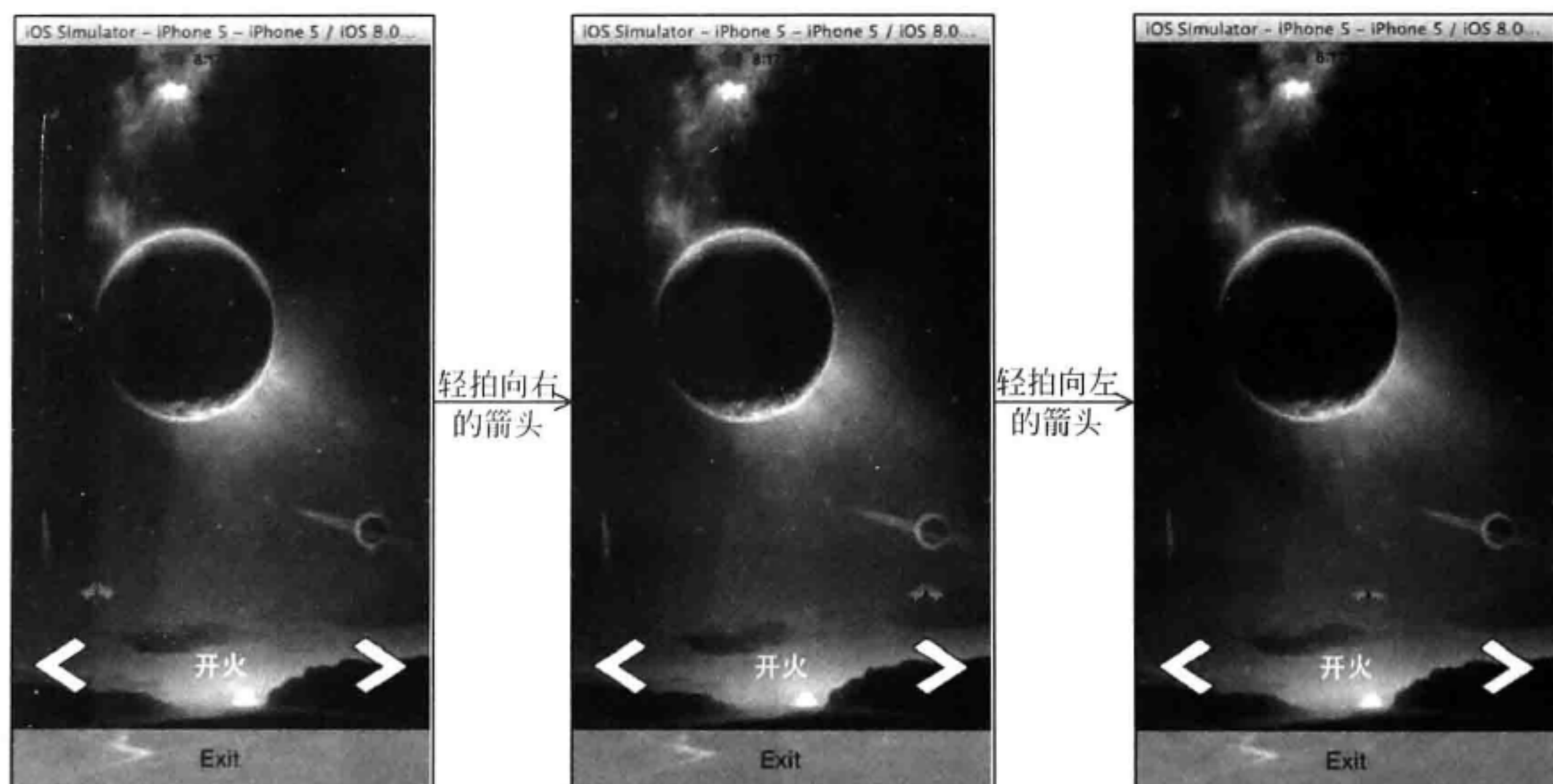


图 6.9 运行效果

6.7 创建敌人

敌人也就是所谓的侵略者,一般都是使用小蜜蜂来代表的。本节将讲解如何去创建一个或者多个敌人。

6.7.1 创建单个敌人的创建

单个敌人的创建是非常简单的,它类似于飞船的添加,也是通过图像视图实现的。在编写代码之前,首先创建一个 Swift File 模板类型的文件,命名为 Enemy。然后在此文件中创建一个基于 NSObject 类的子类 Enemy,该类是用来编写敌人创建的代码的。最后开发者就可以在 Enemy 类中创建敌人了。敌人的创建需要在 initEnemies(gameView:UIView)方法中进行,其代码如下:

```

import Foundation
import UIKit
class Enemy: NSObject {
    var eSize:Int=32
    var gameView:UIView=UIView()
    添加一个敌人
    func initEnemies(gameView:UIView){
        //添加敌人
        self.gameView=gameView
        var enemyImage=UIImage(named: "enemy01.png")
        var enemyView=UIImageView(image: enemyImage)
        enemyView.frame=CGRectMake(CGFloat(10), CGFloat(0), CGFloat(eSize),
            CGFloat(eSize));
        self.gameView.addSubview(enemyView)
    }
}

```

敌人创建好以后，就需要将它显示在界面中，此时需要在 GameViewController 类的 viewDidLoad()方法中进行敌人的添加，即调用 Enemy 类中的 initEnemies(gameView:UIView)方法，代码如下：

```

self.view.addSubview(self.playerView)
var enemy:Enemy=Enemy()
enemy.initEnemies(self.view)

```

此时运行程序，会看到如图 6.10 所示的效果。

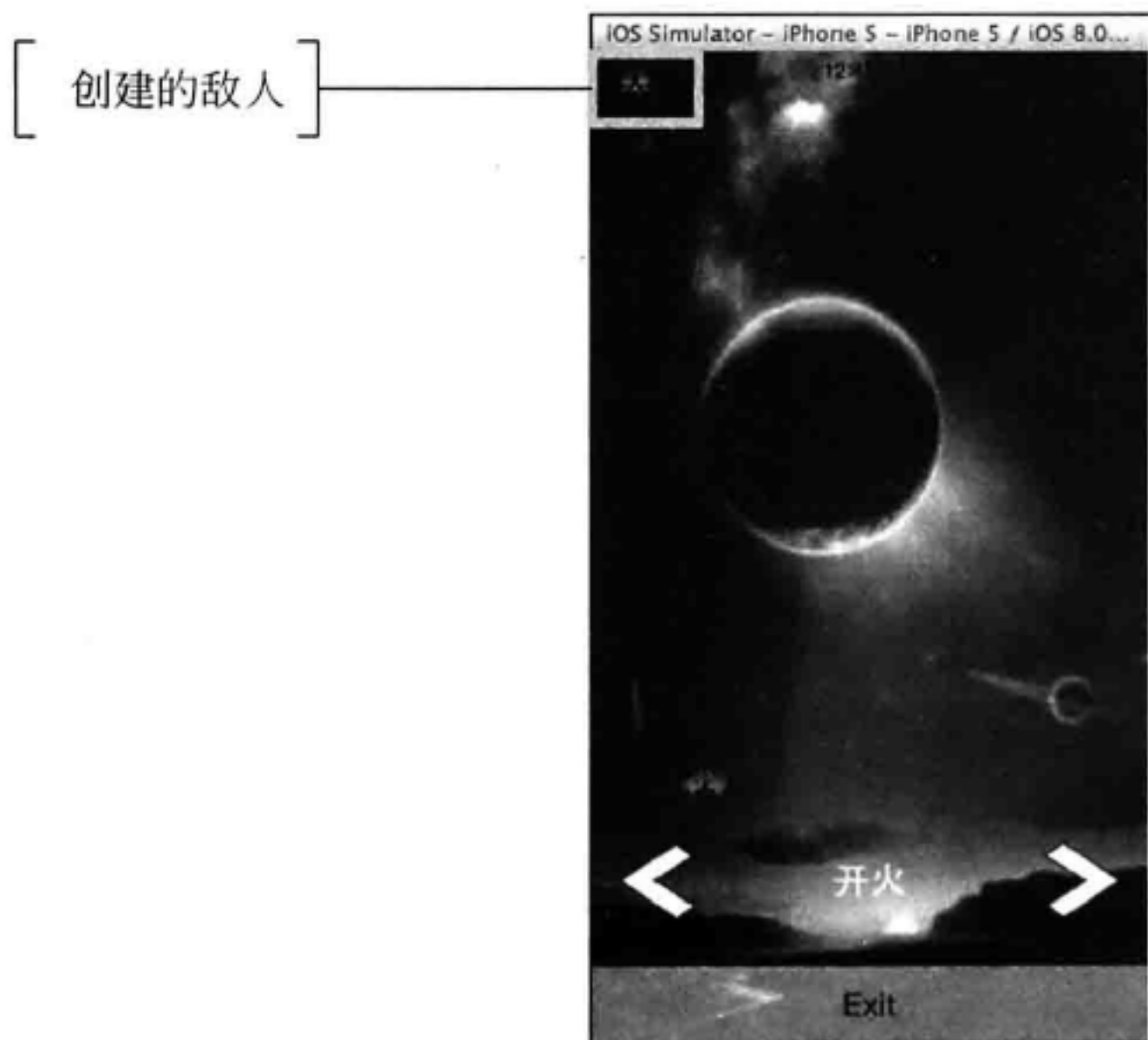


图 6.10 运行效果

6.7.2 创建多个敌人

在太空侵略者的游戏中，会看到敌人不单单只有一个而是有很多个，它们有序的排成了一个矩阵。此时，如果想要我们游戏中的敌人的个数变为多个，开发者可以使用 for 循环语句来实现。代码如下：


```

import Foundation
import UIKit
class Enemy: NSObject {
    var eSize:Int=32
    var gameView:UIView=UIView()
    var enemyList:NSMutableArray=NSMutableArray()
    var enemyRows:Int=5 //设置行数
    var enemyColumns:Int=5 //设置列数
    func initEnemies(gameView:UIView){
        self.gameView=gameView
        var rowCount:Int=0
        var startX:Int=10
        var startY:Int=0
        var enemyImage=UIImage(named: "enemy01.png")
        var i:Int=0
        //创建敌人
        for(i;i<(enemyRows * enemyColumns);i++){
            var columnMod:Int=i%enemyColumns
            if(columnMod==0){
                rowCount++
            }
            //位置的设置
            var xPos:Int=startX+((eSize*columnMod)+(columnMod*5))
            var yPos:Int=startY+((eSize*rowCount)+(rowCount*10))
            //创建一个 enemyView 的图像对象，用来显示敌人
            var enemyView=UIImageView(image: enemyImage)
            enemyView.frame=CGRectMake(CGFloat(xPos),CGFloat(yPos), CGFloat
            (eSize), CGFloat(eSize)); //设置图像视图的框架
            self.enemyList.addObject(enemyView)
            self.gameView.addSubview(enemyView) //添加图像视图对象
        }
    }
}

```

此时运行程序，会看到如图 6.11 所示的效果。

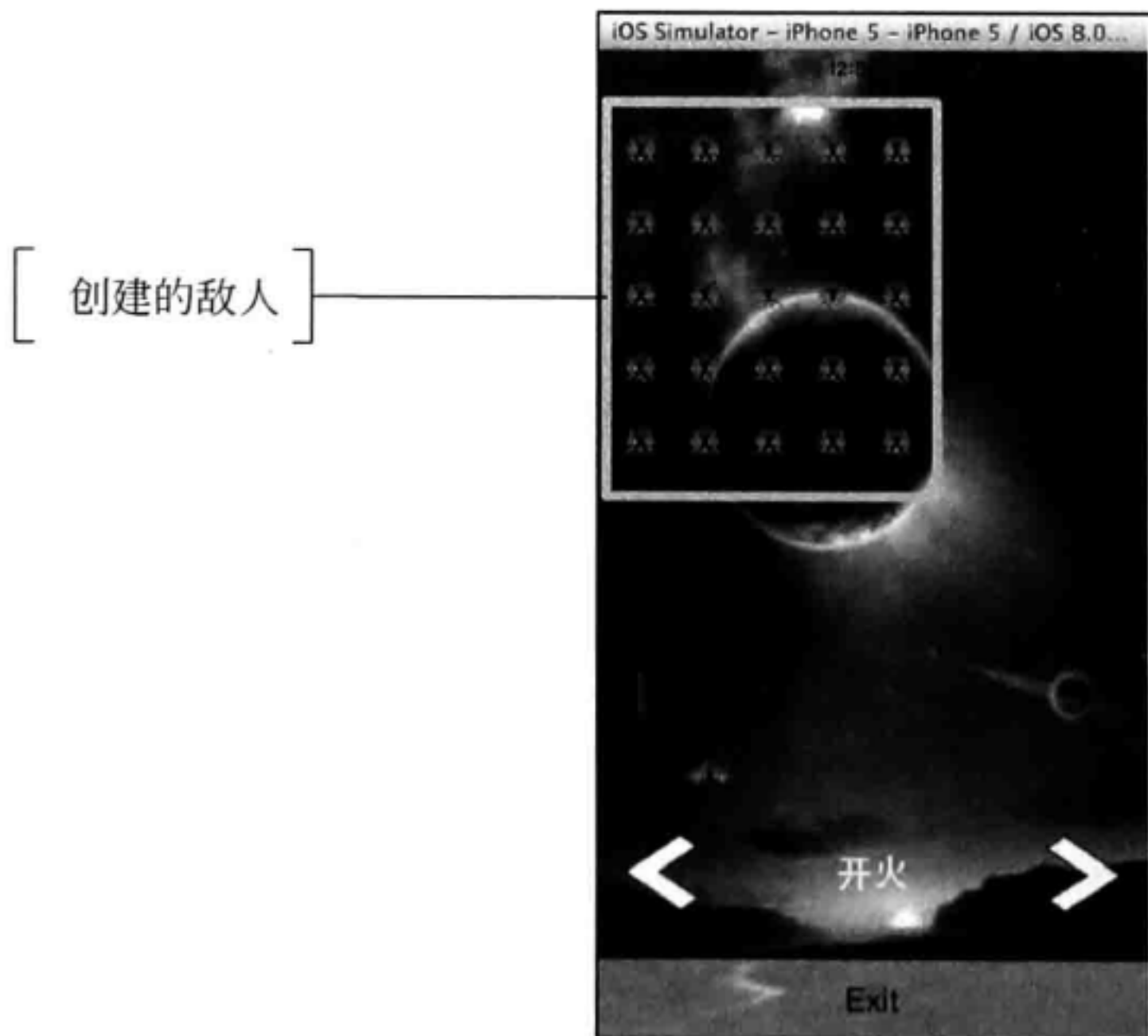


图 6.11 创建的敌人

6.8 移动敌人

在太空侵略者中，敌人是可以自动的进行左右移动的，本节将实现敌人的移动功能。敌人的自动移动也是可以使用定时器实现的。首先需要声明和定义一些变量，代码如下：

```
var enemyTimer:NSTimer=NSTimer()
var minXPos:NSInteger=0
var maxXPos:NSInteger=0
var goingLeft :Bool=false
```

其次在 initEnemies(gameView:UIView)方法中为 minXPos 和 maxXPos 变量赋值。代码如下：

```
self.minXPos=10
self.maxXPos=278
```

最后在 initEnemies(gameView:UIView)方法中实例化一个定时器对象 enemyTimer，代码如下：

```
self.enemyTimer=NSTimer.scheduledTimerWithTimeInterval(0.03, target: self,
selector: Selector("moveEnemies"), userInfo: nil, repeats: true)
```

由于每隔 0.03 秒，定时器就会执行 moveEnemies()方法，此方法实现的功能就是敌人的移动。此方法中的代码如下：

```
func moveEnemies() {
    var enemyView:UIImageView=self.enemyList[0] as UIImageView
    //判断敌人的 x 的位置是否小于 10
    if(Int(enemyView.frame.origin.x) <= self.minXPos){
        goingLeft = false
    }
    enemyView = self.enemyList[enemyColumns-1] as UIImageView
    //判断敌人的 x 的位置是否大于 278
    if(Int(enemyView.frame.origin.x)>=self.maxXPos){
        goingLeft = true
    }
    var i:Int=0
    //遍历，实现敌人的集体移动
    for(i;i<self.enemyList.count; i++){
        enemyView = self.enemyList[i] as UIImageView;
        var xPos:Int=0
        //判断 goingLeft 是否为 true，即敌人的 x 的位置是否于 278
        if(goingLeft){
            xPos = Int(enemyView.frame.origin.x)-3;
        }else{
            xPos = Int(enemyView.frame.origin.x)+3;
        }
        enemyView.frame = CGRectMake(CGFloat(xPos), enemyView.frame.o
rigin.y, CGFloat(eSize), CGFloat(eSize)); //设置敌人的框架
        self.gameView.addSubview(enemyView)
    }
}
```

此时运行程序，会看到如图 6.12 所示的效果。

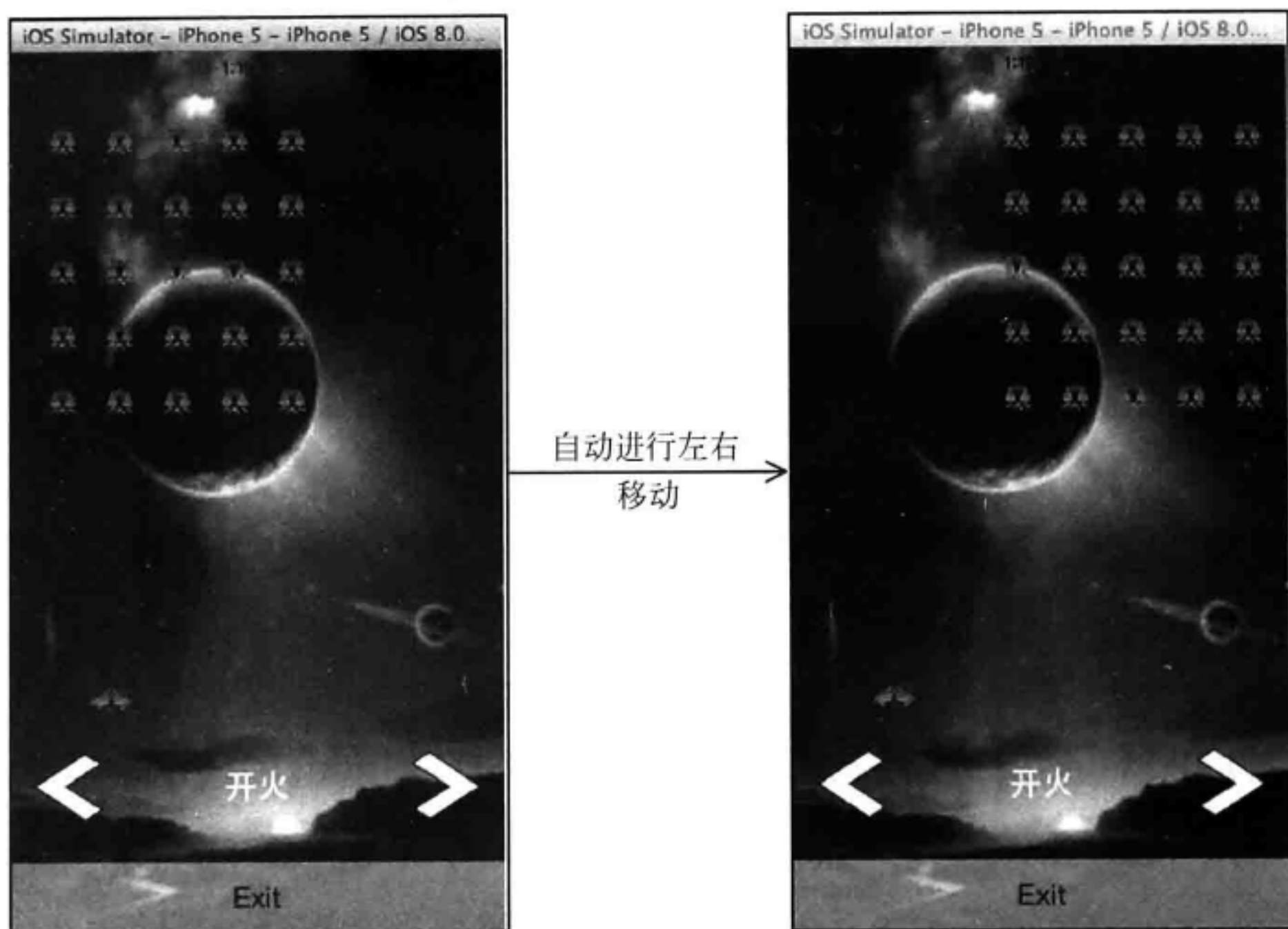


图 6.12 运行效果

6.9 发射子弹

在太空侵略者的游戏中，最重要的就是要实现飞船和敌人的相互战斗。这些战斗主要体现在子弹的发射上面。本节将实现让飞船和敌人发射子弹的功能。

6.9.1 飞船的子弹

对于飞船的发射子弹，开发者可以通过轻拍按钮的功能实现。以下是具体的实现方式。

1. 创建飞船的子弹

飞船发射的子弹也是通过使用图像视图来进行显示的。在实现发射子弹功能之前，首先需要创建一个 Swift File 模板类型的文件，命名为 `PlayerBullet`。然后在此文件中创建一个基于 `NSObject` 类的子类 `PlayerBullet`，该类是用来编写子弹创建的代码。最后开发者就可以在 `PlayerBullet` 类中创建子弹了。其代码如下：

```
import Foundation
import UIKit
class PlayerBullet: NSObject {
    var bulletRect: CGRect = CGRect()
    var bulletView: UIImageView = UIImageView()
    var gameView: UIView = UIView()
    func fireBullet(gameView: UIView, playerView: UIImageView) {
        self.gameView = gameView
        var bombImage1 = UIImage(named: "bullet.png")
    }
}
```



```

        var bulletStartX=playerView.frame.origin.x + (playerView.frame.
            size.width/2) - 4
        var imageRef2=CGImageCreateWithImageInRect (bombImage1.CGImage, CGRectMake
(33, 0, 32, 64))
        self.bulletRect=CGRectMake (bulletStartX,playerView.frame.origin.y,
            8, 16)
        self.bulletView=UIImageView(frame: bulletRect)
        self.bulletView.image=UIImage(CGImage: imageRef2)
        self.gameView.addSubview(self.bulletView)
    }
}

```

⚠注意：显示子弹的图像是图 bullet.png，如图 6.13 所示。

从图 6.13 中可以看到，此时有两个图像，如果要选择其中的一个图作为飞船的子弹，该如何做呢？这时需要使用到 `CGImageCreateWithImageInRect(_image: CGImage!, _rect: CGRect)` 方法，它的功能是截取某一区域的图像，使其形成位图。其语法形式如下：



图 6.13 子弹

```

func CGImageCreateWithImageInRect(_image: CGImage!, _rect: CGRect) ->
CGImage!

```

其中，`_image` 用来指定截取的图像，`_rect` 用来指定截取图像的区域。

2. 为子弹添加动画效果

为了让子弹实现很逼真的效果，开发者可以为子弹添加动画效果。即实现图像 bullet.png 中两个子弹的切换，对于图像的切换可以使用 `UIImageView` 类中的内置动画实现。代码如下：

```

self.bulletView.image=UIImage(CGImage: imageRef2)
self.gameView.addSubview(self.bulletView)
var imageRef=CGImageCreateWithImageInRect (bombImage1.CGImage, CGRectMake
(0, 0, 32, 64))
//实例化 bombArray 可变数组对象
var bombArray:NSMutableArray=NSMutableArray()
bombArray.addObject(UIImage(CGImage: imageRef))
bombArray.addObject(UIImage(CGImage: imageRef2))
self.bulletView.animationImages=bombArray           //加载要实现动画的图像
self.bulletView.animationDuration=0.3                //实现动画持续的时间
self.bulletView.startAnimating()                   //开始动画

```

3. 移动子弹

飞船发射的子弹是要打敌人的，所以子弹要有一个向上移动的动画。对于此动画，我们还是使用定时器进行实现。首先声明一个变量，代码如下：

```

var bulletTimer:NSTimer?=nil

```

然后创建定时器，代码如下：

```

self.bulletTimer=NSTimer.scheduledTimerWithTimeInterval(0.03, target:
self, selector: Selector("moveBullet"), userInfo: nil, repeats: true)

```

在创建的定时器中，可以看到定时器每隔 0.03 秒就会调用一次 `moveBullet()` 方法，在此方法中实现子弹向上移动的功能。代码如下：

```
func moveBullet() {
    self.bulletRect=CGRectOffset(self.bulletRect, 0, -5)
    self.bulletView.frame=self.bulletRect
    //判断子弹的 y 的位置是否小于 50
    if(self.bulletRect.origin.y<50){
        self.bulletTimer?.invalidate()
        self.bulletTimer=nil
        self.bulletView.removeFromSuperview() //移除子弹
    }
}
```

4. 让飞船发射子弹

最后就是轻拍“开火”按钮实现飞船发射子弹的功能了。首先将“开火”按钮关联一个 `fireButton()` 的动作。当玩家轻拍此按钮后，就可以触发此动作。关联好动作后，就可以实现飞船开火的效果了，代码如下：

```
@IBAction func fireButton(sender: AnyObject) {
    var fire:PlayerBullet=PlayerBullet()
    fire.fireBullet(self.view, playerView: self.playerView)
}
```

此时运行程序，会看到如图 6.14 所示的效果。

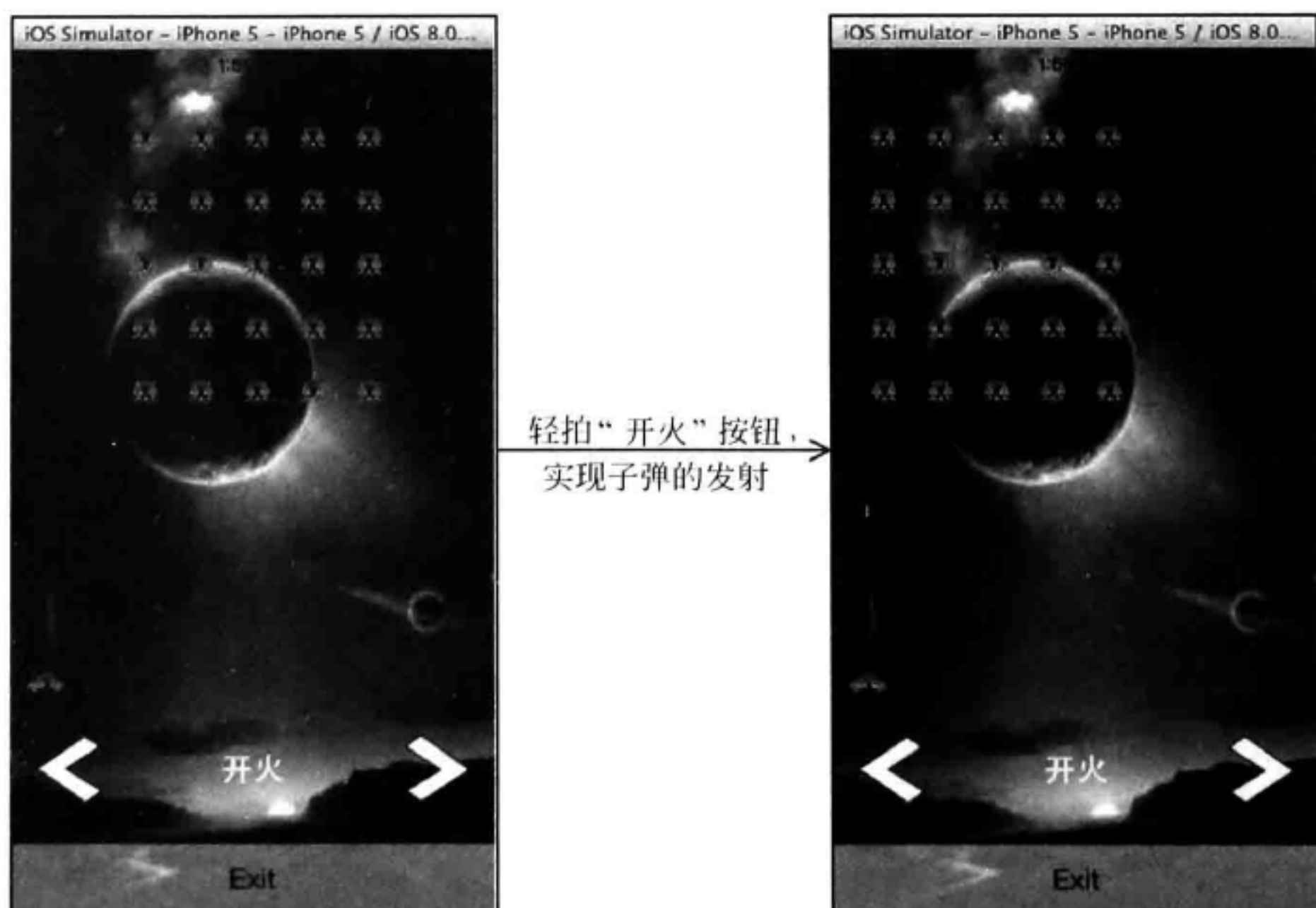


图 6.14 运行效果

6.9.2 敌人的子弹

飞船发射子弹的功能实现后，再来实现敌人发射子弹的效果。具有步骤如下所述。

1. 创建子弹

敌人的子弹的创建和飞船的子弹的创建是一样的。首先需要创建一个 Swift File 模板类型的文件，命名为 `EnemyBullet`。然后在此文件中创建一个基于 `NSObject` 类的子类 `EnemyBullet`，该类是用来编写子弹创建的代码的。最后开发者就可以在 `EnemyBullet` 类中创建子弹了。其代码如下：

```
import Foundation
import UIKit
class EnemyBullet: NSObject {
    var bombRect:CGRect=CGRectMake(0, 0, 0, 0)
    var bombView:UIImageView=UIImageView()
    var gameView:UIView=UIView()
    func fireBullet(gameView:UIView,enemyList:NSArray){
        var randEnemy:Int=random()%enemyList.count
        var enemyView:AnyObject=enemyList[randEnemy]
        self.gameView=gameView
        var bombImage1:UIImage=UIImage(named: "bullet.png")
        var bombStartX=enemyView.frame.origin.x
        //截图
        var imageRef=CGImageCreateWithImageInRect(bombImage1.CGImage,
            CGRectMake(0, 0, 32, 64))
        var imageRef2=CGImageCreateWithImageInRect(bombImage1.CGImage,
            CGRectMake(33, 0, 32, 64))
        //实例化 bombArray 可变数组对象，并为此对象添加对象
        var bombArray:NSMutableArray=NSMutableArray()
        bombArray.addObject(UIImage(CGImage: imageRef))
        bombArray.addObject(UIImage(CGImage: imageRef2))
        self.bombRect=CGRectMake(bombStartX, enemyView.frame.origin.y, 8, 16)
        self.bombView.frame=self.bombRect
        self.bombView.image=UIImage(CGImage: imageRef2)
        //实现图像的动画效果
        self.bombView.animationImages=bombArray
        self.bombView.animationDuration=3
        self.gameView.addSubview(self.bombView)
        self.bombView.startAnimating()
    }
}
```

2. 移动子弹

敌人发射的子弹是用来攻打飞船的。由于飞船在敌人的下方，所以子弹需要有一个向下的动画效果，此动画，我们还是使用定时器进行实现。首先声明一个变量，代码如下：

```
var bombTimer:NSTimer?=nil
```

然后创建定时器，代码如下：

```
self.bombTimer=NSTimer.scheduledTimerWithTimeInterval(0.03, target: self,
    selector: Selector("moveBomb"), userInfo: nil, repeats: true)
//实现子弹的移动
func moveBomb(){
    self.bombRect=CGRectOffset(self.bombRect, 0, 5)
    self.bombView.frame=self.bombRect
    //判断子弹的 y 的位置是否大于 440
    if(self.bombRect.origin.y>440){
        self.bombTimer?.invalidate()
        self.bombTimer=nil
    }
}
```



```

        self.bombView.removeFromSuperview()
    }
}

```

3. 让敌人发射子弹

最后打开 Enemy.swift 文件, 编写代码, 实现敌人的子弹的发射。由于敌人的子弹不需要玩家进行控制, 而是自动进行发射的, 所以还需要使用定时器实现发射的动画效果。首先, 声明一个变量, 代码如下:

```
var enemyBulletTimer: NSTimer = NSTimer()
```

然后创建定时器, 代码如下:

```
self.enemyBulletTimer = NSTimer.scheduledTimerWithTimeInterval(1, target: self, selector: Selector("dropBomb"), userInfo: nil, repeats: true)
```

在创建的定时器中, 可以看到定时器每隔 1 秒就会调用一次 dropBomb() 方法, 在此方法中实现子弹的发射的功能。代码如下:

```

func dropBomb() {
    var newBullet: EnemyBullet = EnemyBullet()
    newBullet.fireBullet(self.gameView, enemyList: self.enemyList)
}

```

此时运行程序, 会看到如图 6.15 所示的运行效果。

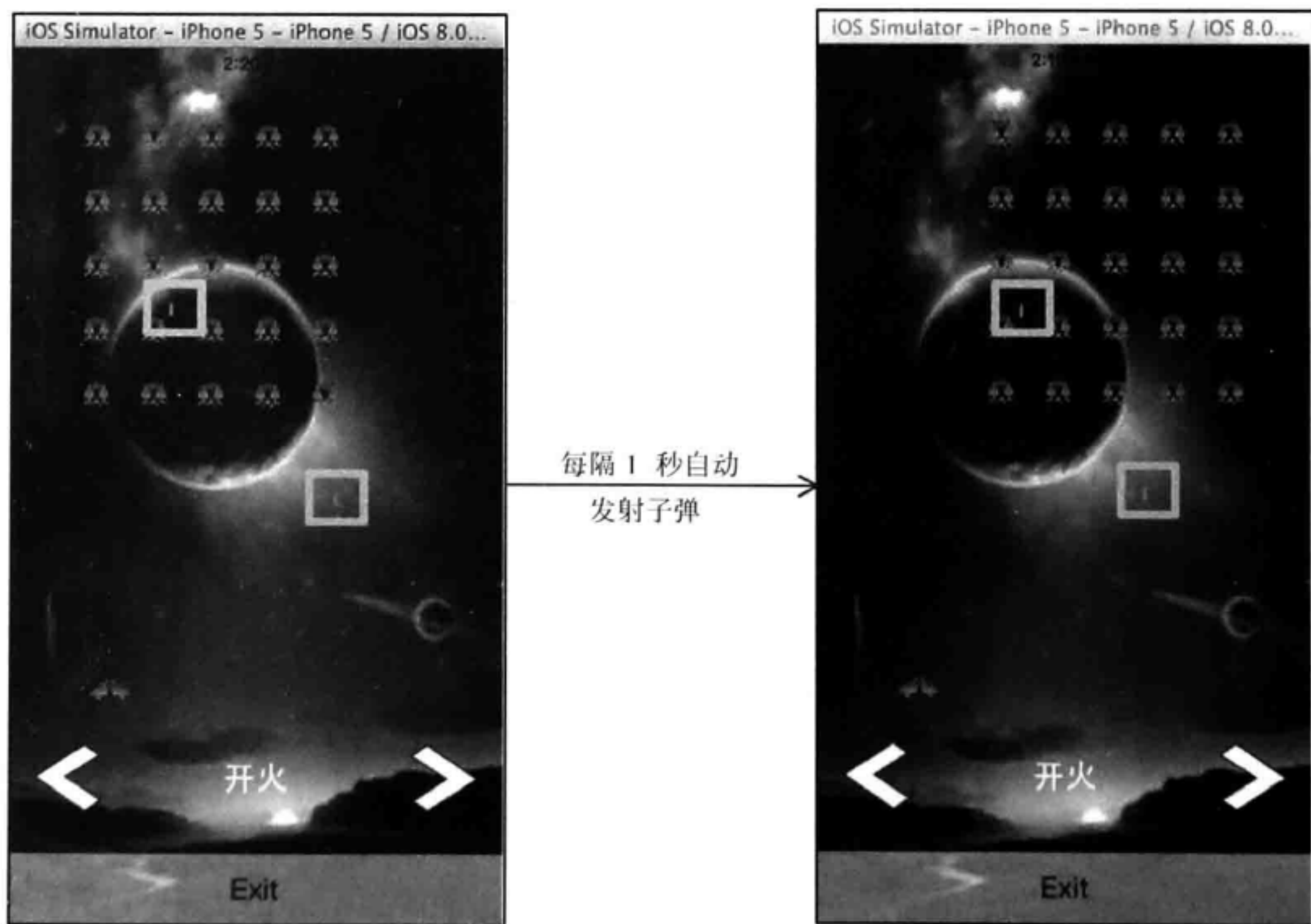


图 6.15 运行效果

⚠注意: 在图中用浅颜色标出的地方就是敌人发射的子弹。

6.10 场景的切换

最后就是要将主菜单的界面和射击游戏的界面进行切换，具体操作步骤如下所述。

(1) 打开 Main.storyboard 文件。

(2) 单击实现主菜单的视图控制器，在此视图控制器中，选择界面上方的 Dock 中的 View Controller 图标，在工具窗口中的 Show the Attributes inspector 选项，即属性查看器中，找到 Is Initial View Controller 复选框，将其选中，使此视图控制器成为初始视图控制器。

(3) 按住 Ctrl 键拖动 View Controller 视图控制器的界面中的 Play Game 按钮对象到 Game View Controller 视图控制器的界面中。

(4) 松开鼠标后，会弹出一个 Action Segue 对话框，选择其中的 modal 选项，此时就实现了主菜单的界面到射击游戏界面的切换，此时的画布效果如图 6.16 所示。



图 6.16 画布的效果

(5) 按住 Ctrl 键拖动 Game View Controller 视图控制器的界面中的 Exit 按钮对象到 View Controller 视图控制器的界面中。

(6) 松开鼠标后，会弹出一个 Action Segue 对话框，选择其中的 modal 选项，此时就实现了射击游戏界面到主菜单界面的切换。此时的画布效果如图 6.17 所示。

此时运行程序，会看到如图 6.18 所示的效果。

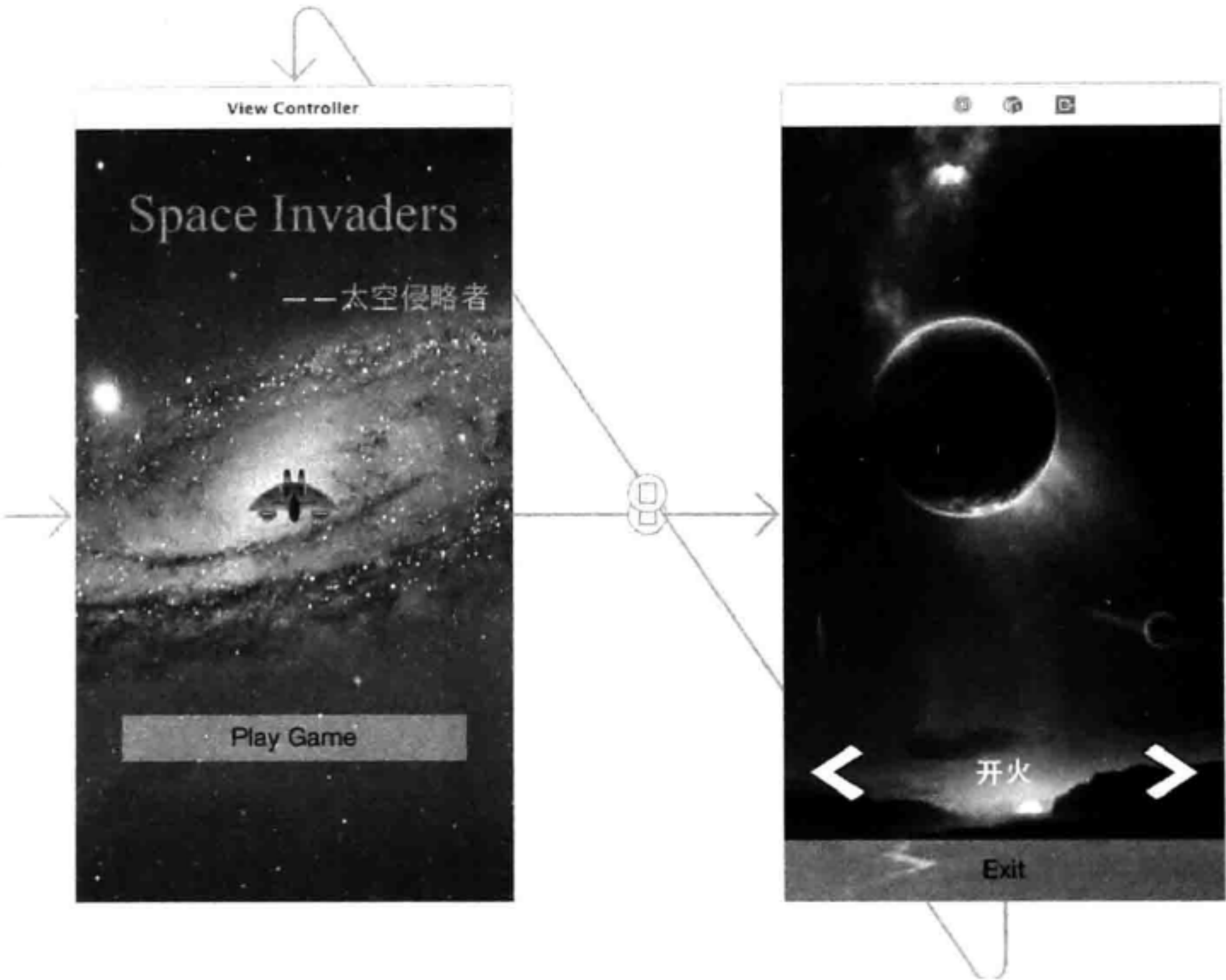


图 6.17 画布的效果

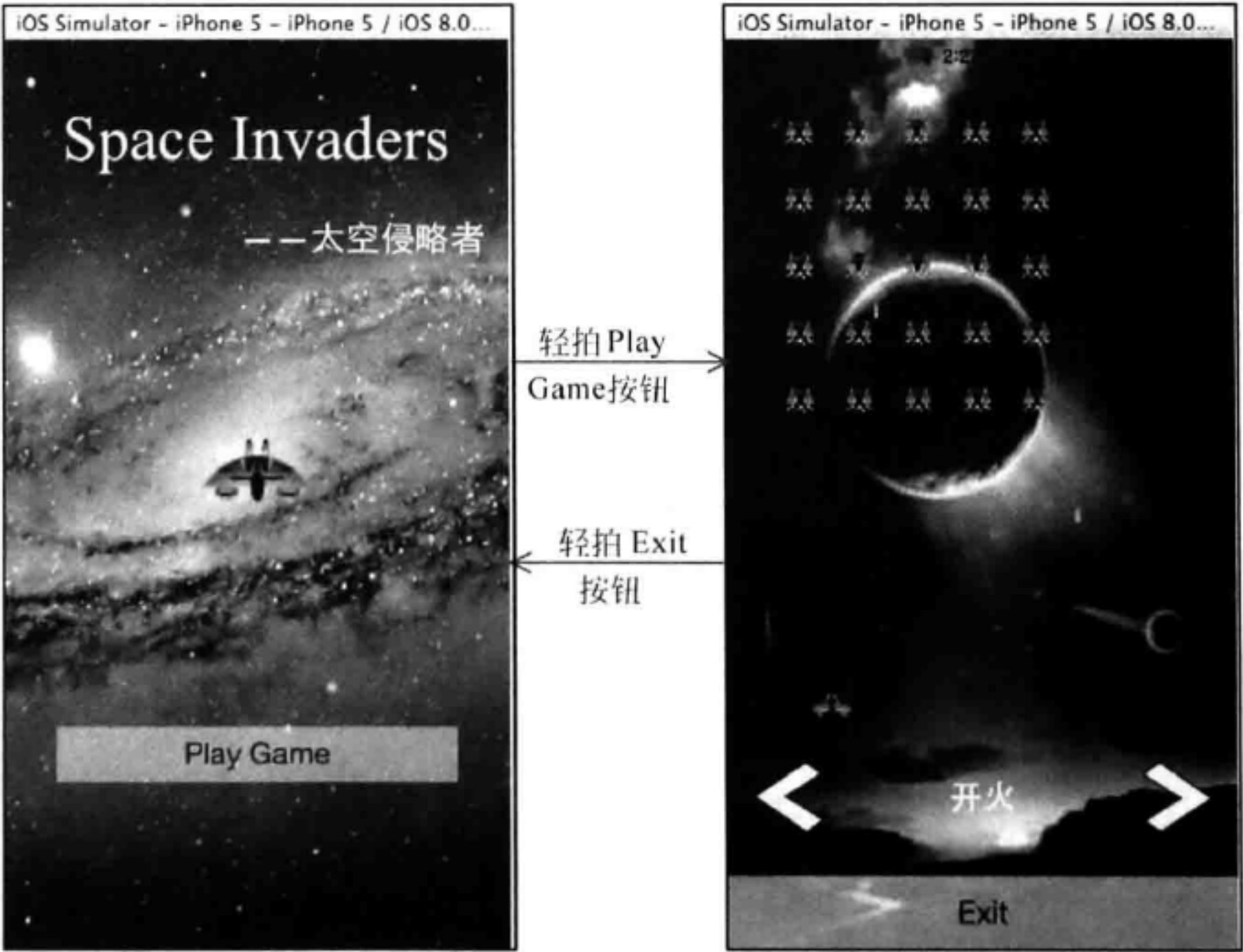


图 6.18 运行效果

第 7 章 太空侵略者 2——游戏引擎

在上一章中讲解了太空侵略者游戏的实现。但是在这个游戏中，敌人的子弹击中飞船或者是飞船的子弹击中敌人都是没有任何反应的。本章将在上一章游戏的基础上为游戏做一些改进，如使敌人的子弹可以击中飞船，或者飞船的子弹可以击中敌人等。

7.1 游戏介绍

本节我们将为游戏的各个场景添加很多的内容。例如，在主菜单中添加一个新的按钮，此按钮的标题为 Score List。轻拍它之后可以打开一个新的场景，即分数榜的界面等。以下就是对这些新增内容的介绍。

1. 主菜单

在主菜单中添加一个新的按钮，此按钮的标题为 Score List。它的功能是轻拍之后可以打开一个新的场景，即分数榜的界面，如图 7.1 所示。

2. 射击游戏界面

在此界面中添加一个标签，在此标签中可以显示分数，如图 7.2 所示。



图 7.1 主菜单界面



图 7.2 游戏界面

3. 分数榜的界面

分数榜的界面是一个新增的界面，它可以用来显示玩家的一些分数，如图 7.3 所示。



图 7.3 分数榜

7.2 开发游戏前的准备工作

在开发太空侵略者 2 游戏之前，需要做一些准备工作，这些准备工作如下所述。

1. 创建工程

在制作游戏之前，首先需要创建一个 Single View Application 模板类型的项目，将其命名为 Space Invaders2。

2. 添加图像

添加图像 backdrop.jpg、background.jpg、background1、bullet.png、enemy01.png、left.png、loading.png、right.png 和 ship.png 到创建项目的 Supporting Files 文件夹中，如图 7.4 所示。

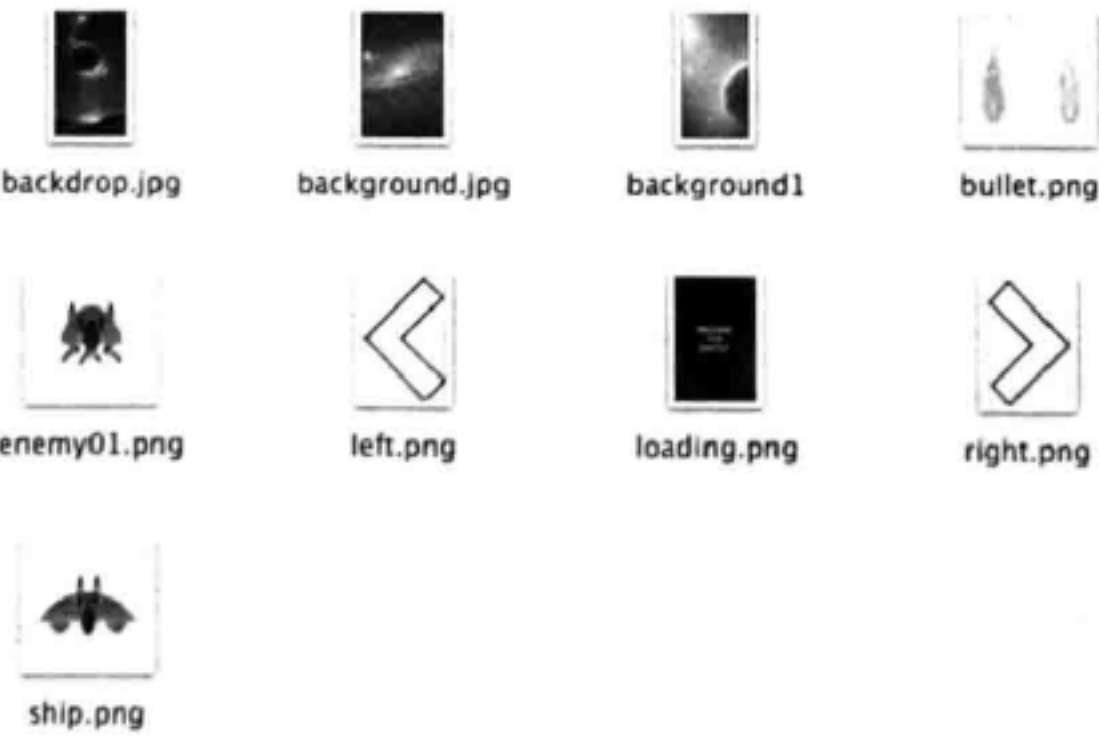


图 7.4 添加的图像

7.3 主菜单模块

主菜单的界面和在上一章中介绍的界面有一点不同，就是多添加了一个按钮。此按钮的功能是轻拍后实现进入排行榜的界面。主菜单界面的效果如图 7.5 所示。

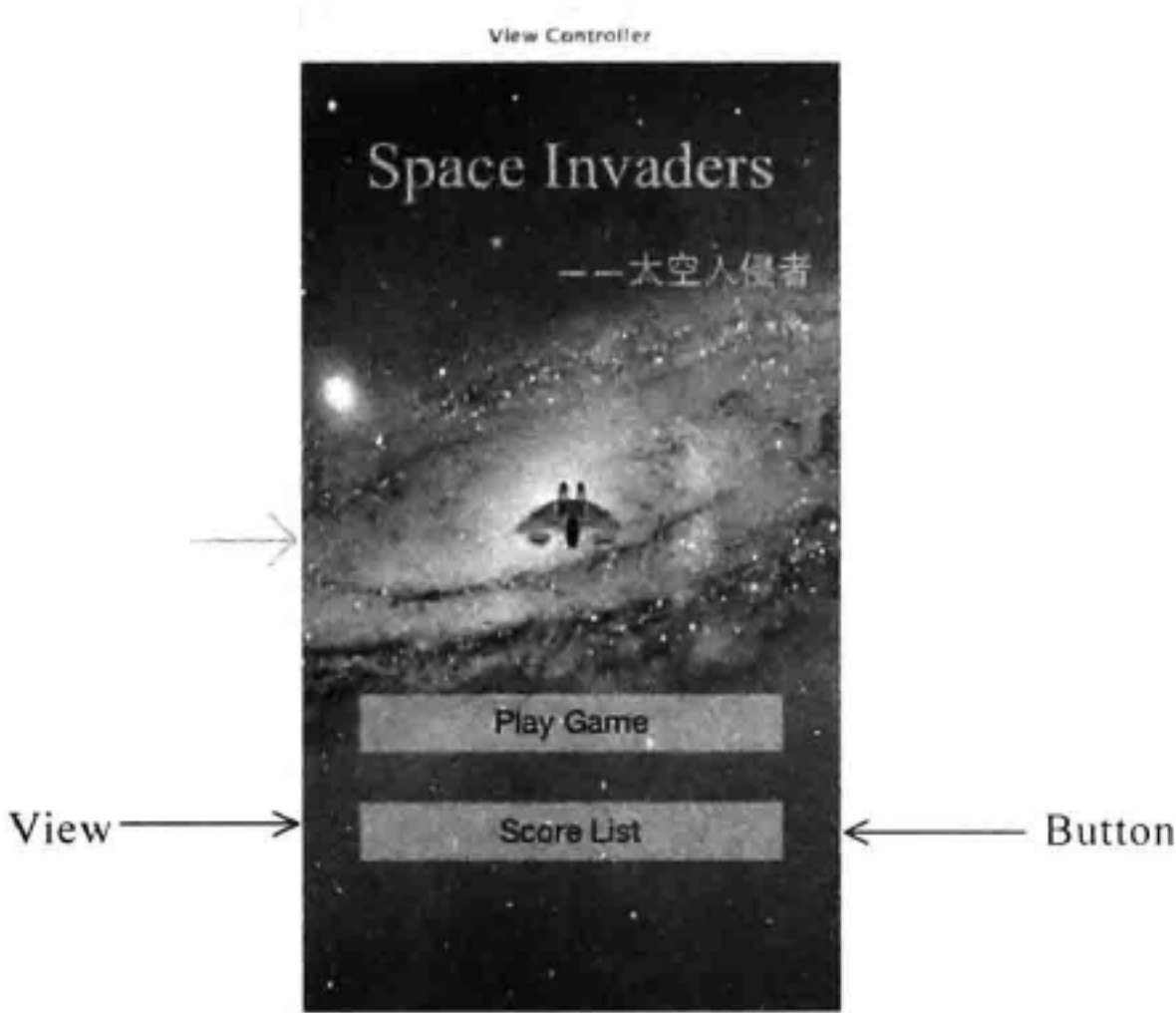


图 7.5 界面的效果

需要添加的视图对象，以及对它们的设置，如表 7-1 所示。

表 7-1 设置界面

视 图	设 置
View	Alpha: 0.4 位置和大小: (34, 442, 252, 34)
Button	Title: Score List Font: System 19.0 Text Color: 黑色 位置和大小: (35, 2, 182, 29)

注意：在对主菜单的界面设置好以后，打开 Show the Identity inspector 面板，即标识查看器。在其中找到 Storyboard ID，在此输入框中输入 back（这个 back 表示一个标识符，在后面会使用到）。然后，将 Use Storyboaed ID 单选框选中，如图 7.6 所示。

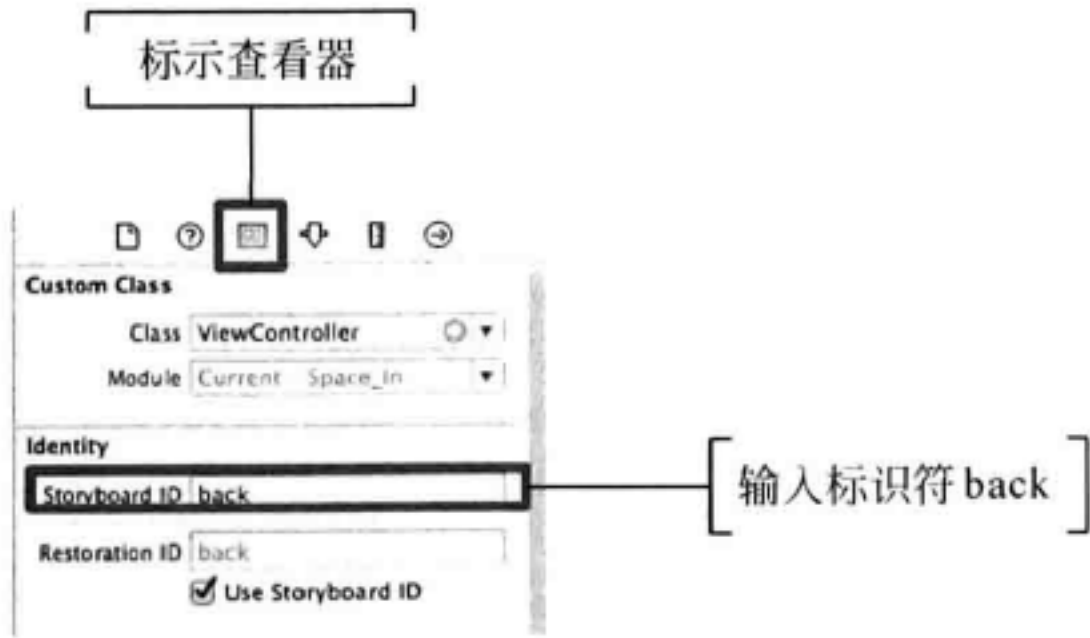


图 7.6 设置

7.4 射击游戏模块

在此游戏中的射击游戏界面和第 6 章中的射击游戏界面是一样的，唯一不同的是，在此游戏的射击游戏界面中添加了两个标签，用来显示飞船射击的分数，如图 7.7 所示。

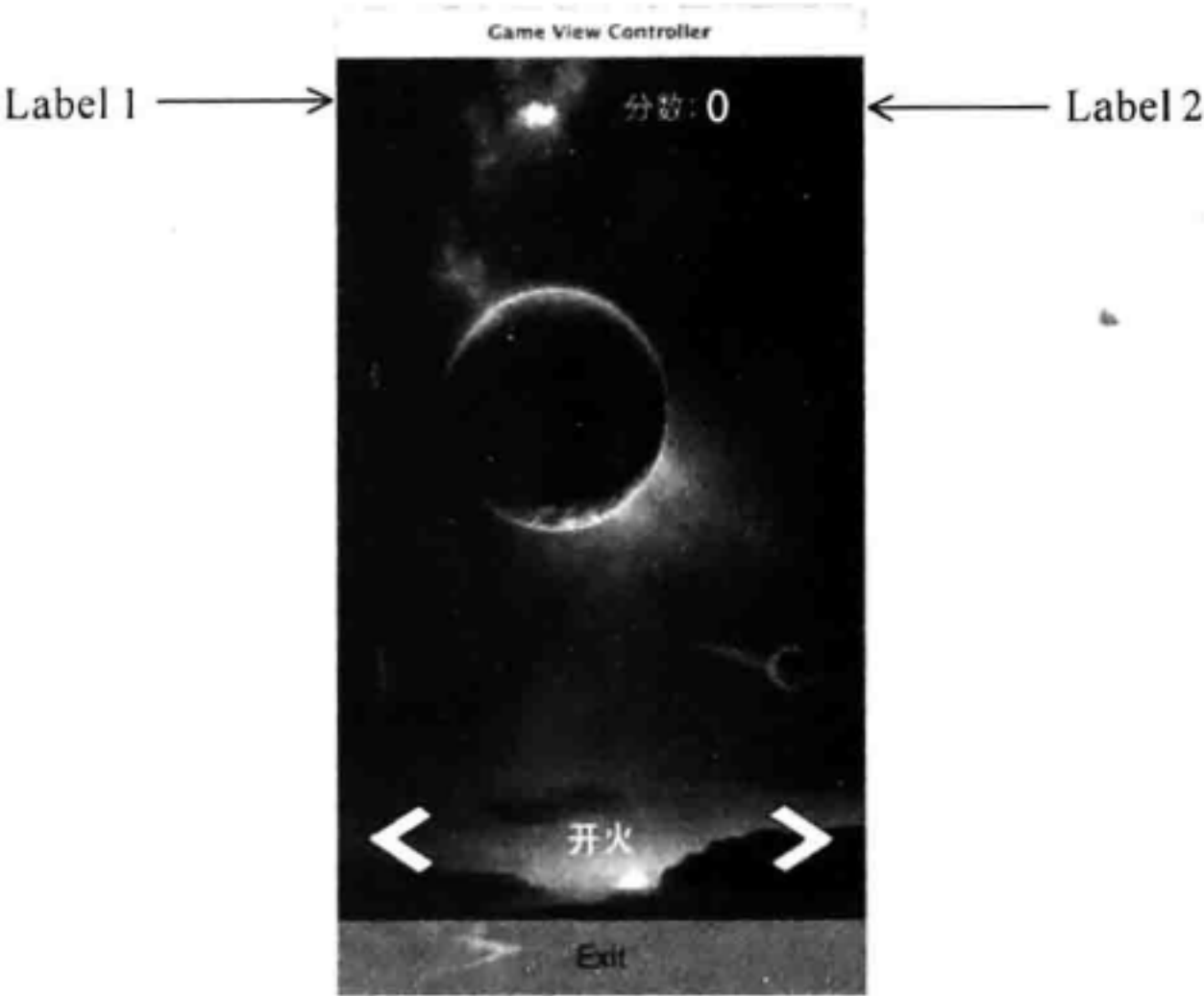


图 7.7 界面效果

需要添加的视图对象，以及对它们的设置，如表 7-2 所示。

表 7-2 设置界面

视 图	设 置
Label1	Text: 分数: Color: 白色 Font: System 19.0 位置和大小: (174, 20, 61, 21)
Label2	Text: 0 Color: 白色 Font: System 19.0 Alignment: 居中 位置和大小: (224, 20, 80, 21)

注意：在设计射击游戏界面时，首先需要创建一个 Swift File 模板类型的文件，命名为 Game。然后在此文件中创建一个基于 UIViewController 类的子类 GameViewController，该类是用来编写对射击游戏界面的一些操作。单击作为射击游戏界面的视图控制器，选择界面上方的 Dock 中的 View Controller 图标。在工具窗口中的 Show the Identity inspector 选项，即标示查看器中，将 Custom Class 下的 Class 设置为创建的 GameViewController 类。这时在画布中的这个视图控制器就变为了 Game View Controller 视图控制器。

7.5 了解状态机

状态机简称为 FSM (Finite State Machine)，也可以称为有限状态机。状态机是状态转移图，即在状态不同的条件下跳转到自己或者不同状态的图中。例如，在一个游戏中一般有两种状态：处于玩游戏的状态和结束游戏的状态。触发的条件有两个，轻拍开始游戏的按钮和敌人打死玩家的角色。所以，在游戏中的状态机就有两种形式，如图 7.8 和图 7.9 所示。



图 7.8 状态机

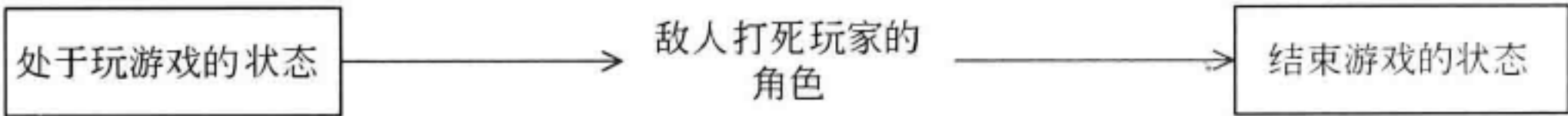


图 7.9 状态机

状态机的使用不仅便于阅读、理解和维护，更重要的是利于综合器优化代码，利于开发者添加合适的约束条件等。状态机一般可以使用 switch/case 或者 if/else 来实现。在本章所讲的游戏中就使用到了状态机，因为本章的游戏会处于多个状态，如初始状态、玩游戏的状态等。它的实现方式如下所述。

首先创建一个枚举类型，在枚举类型中放置的成员就是本章游戏所处的 4 个状态。代码如下：

```
enum GAME_STATE:Int{
    case INITIALIZING=1
    case PLAYING=2
    case RELOADING=3
    case ENDING=4
}
```

然后，实例化一个枚举对象，代码如下：

```
var currentState=GAME_STATE.INITIALIZING
```

最后使用 switch/case 的方法实现状态机。代码如下：

```
func changeState(newState:GAME_STATE) {
    self.currentState=newState
    switch(newState) {
        //初始化的状态
        case .INITIALIZING:
            self.loadingScreen()
        //玩的状态
        case .PLAYING:
            break
        //重写进入游戏界面的状态
```

```

case .RELOADING:
    self.view.addSubview(self.loadingView)
    self.enemies.stopTimers()
    self.collisionTimer?.invalidate()
    self.collisionTimer=nil
//结束游戏的状态
case .ENDING:
    break
//其他
default:
    var alert=UIAlertView()
    alert.title="Integer out of range"
    alert.addButtonWithTitle("Cancel")
    alert.show()
}
}

```

7.6 使用代码添加射击游戏界面元素

此时的射击游戏界面是一个只有按钮的界面，没有飞船、敌人和子弹这些物体。本节将讲解这些物体的添加。

7.6.1 提示界面

所谓提醒界面，就是在单击主菜单的开始游戏按钮后，随后弹出的界面。它主要的作用是提醒玩家即将进入游戏，也可以用来告诉玩家此游戏的玩法，或者是游戏的一些规则等。例如在经典的黄金矿工游戏中，当玩家打开此游戏后，首先进入的就是游戏的主菜单，如图 7.10 所示。



图 7.10 主菜单界面

当玩家轻拍主菜单中的 Start 按钮后，就会进入游戏的提示界面，在此界面中，就是告诉玩家此游戏的规则是什么，即提示界面告诉玩家首次目标是\$650，如图 7.11 所示。在此提示界面上停留 2~3 秒后，就会进入游戏的界面，如图 7.12 所示。



图 7.11 提示界面

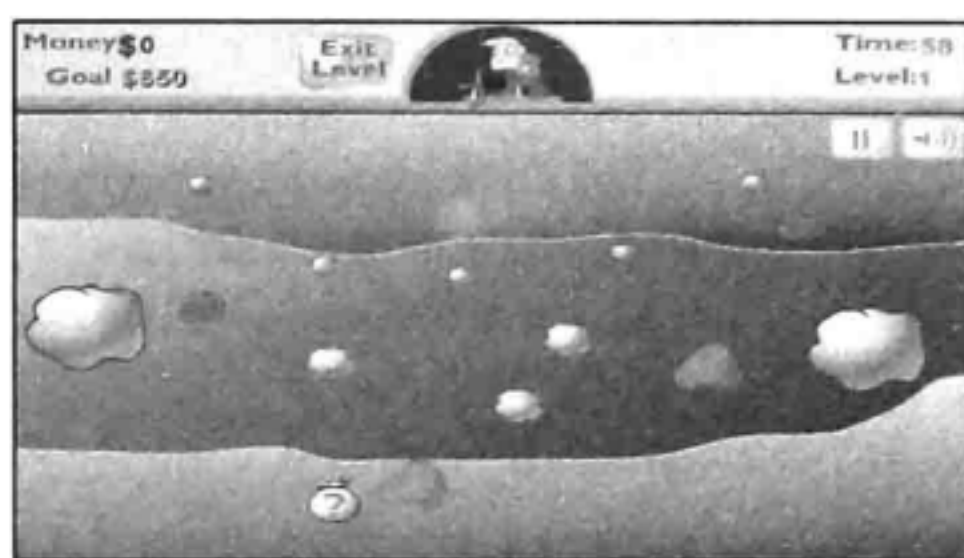


图 7.12 游戏界面

在本章的游戏中，为了让玩家可以调整好心情做好战斗的准备，我们也可以在打开射击游戏界面前，添加一个提示界面。此提示界面使用 Image View 图像视图来实现。打开 Game.swift 文件，编写代码，实现提示界面的添加，代码如下：

```
import Foundation
import UIKit
class GameViewController: UIViewController {
    var loadingView:UIImageView=UIImageView()
    func loadingScreen(){
        //添加提示界面
        self.loadingView.image=UIImage(named: "loading.png")
        self.loadingView.frame=self.view.frame
        self.view.addSubview(self.loadingView)
        NSTimer.scheduledTimerWithTimeInterval(2, target: self, selector:
        Selector("closeScreen"), userInfo: nil, repeats: false)//创建定时器
    }
    //关闭提示界面
    func closeScreen(){
        self.loadingView.removeFromSuperview() //从主视图中移除视图对象
    }
    //视图在加载后调用的方法
    override func viewDidLoad() {
        super.viewDidLoad()
        self.changeState(.INITIALIZING)
    }
}
```

此时，运行程序会看到如图 7.13 所示的效果。

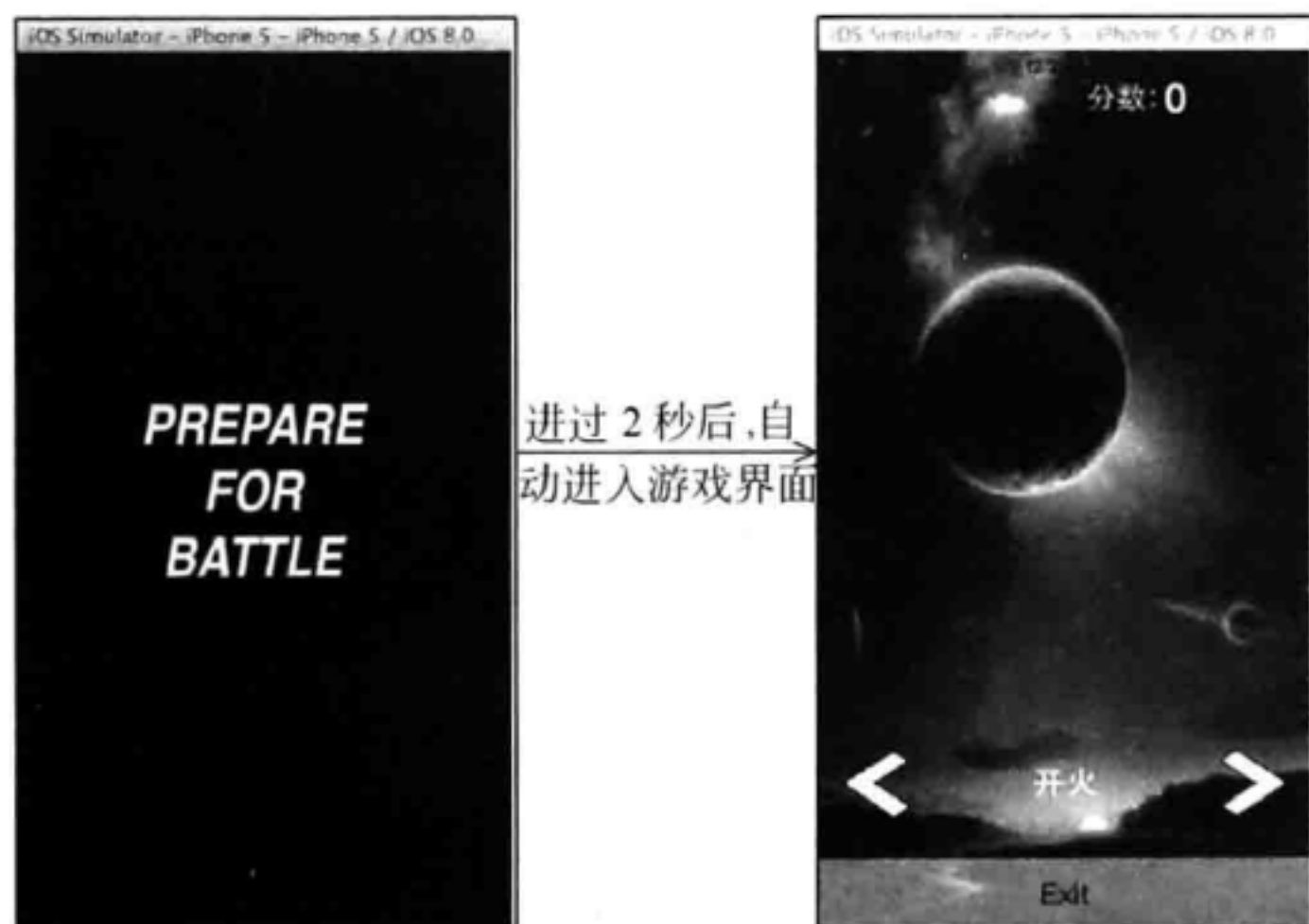


图 7.13 运行效果

🔔注意：在运行程序前，首先需要回到 Main.storyboard 文件中。单击 Game View Controller 视图控制器。在此视图控制器中，选择界面上方的 Dock 中的 Game View Controller 图标。在工具窗口中的 Show the Attributes inspector 选项，即属性查看器中，找到 Is Initial View Controller 复选框，将其选中。此时 Game View Controller 视图控制器就成为了初始视图控制器。

7.6.2 飞船

在上一章中已经讲解过了飞船代表正义的一方，本小节将讲解有关飞船的一些操作。

1. 创建飞船

与上一章不同，在本章中，我们将对于飞船的操作都放入一个文件中以便于管理。所以在编写对飞船操作的代码时，首先需要创建一个 Swift File 模板类型的文件，命名为 PlayerObject。然后在此文件中创建一个基于 NSObject 类的子类 PlayerObject，该类是用来编写对飞船的一些操作的。首先就是创建飞船，在本章中创建飞船的方式和在上一章中的方式是相同的，都是使用代码来创建。代码如下：

```
import Foundation
import UIKit
class PlayerObject: NSObject {
    var gameView:UIView=UIView()
    var playerRect:CGRect=CGRect()
    var playerImage:UIImage=UIImage(named: "ship.png")
    var playerView:UIImageView=UIImageView()
    func initPlayer(gameView:UIView)->PlayerObject{
        self.gameView=gameView
        self.playerView.image=self.playerImage //设置图像视图显示的图像
        self.playerRect=CGRectMake(50, 400, 32, 32)
        self.playerView.frame=self.playerRect //设置图像视图的框架
        self.gameView.addSubview(self.playerView) //添加图像视图
        return self
    }
}
```

此时飞船就创建完成了。以下实现的功能是在射击游戏的界面中显示飞船，首先需要 PlayerObject.swift 文件中添加一个 getPlayerView()的方法，在此方法中实现图像视图的获取。代码如下：

```
func getPlayerView()->UIView{
    return self.playerView
}
```

然后在 Game.swift 文件中声明两个变量，代码如下：

```
var playerView:UIView=UIView()
var playerOne:PlayerObject=PlayerObject() //实现飞船的对象
```

最后，在 loadingScreen()方法中，将飞船进行显示，代码如下：


```

playerOne=PlayerObject().initPlayer(self.view)
self.playerView=self.playerOne.getPlayerView()
self.loadingView.image=UIImage(named: "loading.png")

```

此时运行程序，开发者就会看到在射击游戏的界面会出现一个小小的飞船，此飞行的位置为（50, 400），大小为（32, 32），如图 7.14 所示。

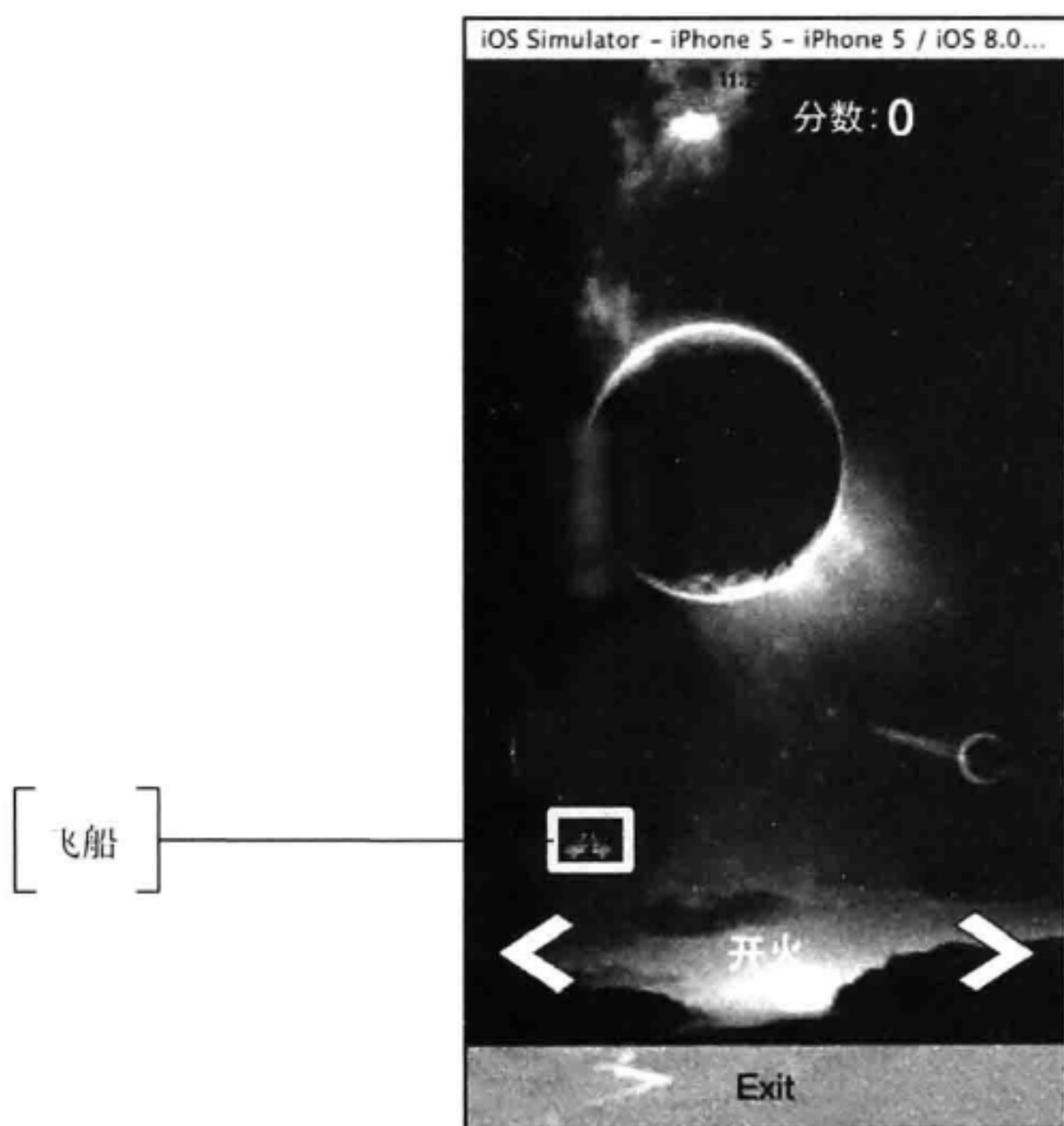


图 7.14 运行效果

2. 移动飞船

本章中飞船的移动也和上一章中所讲的飞船移动的实现方式一致。首先，在 PlayerObject.swift 文件中添加一个变量，此变量是关于定时器的。代码如下：

```
var moveTimer:NSTimer?=NSTimer()
```

然后添加 movePlayerRight()方法和 movePlayerLeft()方法实现飞船的向左和向右移动。代码如下：

```

//向右移动
func movePlayerRight() {
    var screenRect:CGRect=UIScreen.mainScreen().bounds
    //判断飞船的 x 的位置是否超出边界
    if(self.playerRect.origin.x<=screenRect.size.width-self.playerRect.
        size.width - 10){
        self.playerRect = CGRectOffset(self.playerRect, 3, 0);
        self.playerView.frame = self.playerRect;
    }
}
//向左移动

```



```
func movePlayerLeft() {
    //判断飞船的 x 的位置是否超出边界
    if(self.playerRect.origin.x >= 10){
        self.playerRect = CGRectOffset(self.playerRect, -3, 0);
        self.playerView.frame = self.playerRect;
    }
}
```

打开 Game.swift 文件，将此文件和射击游戏界面中的向左的按钮进行动作 moveLeft() 的关联。当玩家轻拍向左的按钮后，就可以触发此动作。关联好动作后，就可以实现飞船的向左移动动画了，此时的动画需要使用定时器实现，所以需要在 moveLeft() 动作中添加以下的代码：

```
@IBAction func moveLeft(sender: AnyObject) {
    self.releaseTouch()
    //判断当前的状态是否为 PLAYING
    if(self.currentState != .PLAYING){
        return
    }
    moveTimer=NSTimer.scheduledTimerWithTimeInterval(0.03, target: self.
playerOne, selector: Selector("movePlayerLeft"), userInfo: nil, repeats:
true)
}
```

将 Game.swift 文件和射击游戏界面中的向右的按钮进行动作 moveRight() 的关联，当玩家轻拍向右的按钮后，就可以触发此动作。关联好动作后，就可以实现飞船的向右移动动画了，此时的动画也需要使用定时器实现，所以需要在 moveRight() 动作中添加以下的代码：

```
@IBAction func moveRight(sender: AnyObject) {
    self.releaseTouch()
    //判断当前的状态是否为 PLAYING
    if(self.currentState != .PLAYING){
        return
    }
    moveTimer=NSTimer.scheduledTimerWithTimeInterval(0.03, target: self.
playerOne, selector: Selector("movePlayerRight"), userInfo: nil, repeats:
true)
}
```

最后需要添加一个 releaseTouch() 方法。此方式实现的功能是让 moveTimer 定时器失效，这一点也在上一章中讲解过了。代码如下：

```
func releaseTouch() {
    //判断定时器是否为空
    if(self.moveTimer != nil){
        self.moveTimer?.invalidate()
        self.moveTimer = nil;
    }
}
```

此时运行程序可以看到以下的效果。当玩家轻拍向右的按钮后，飞船会向右移动，如图 7.15 所示。

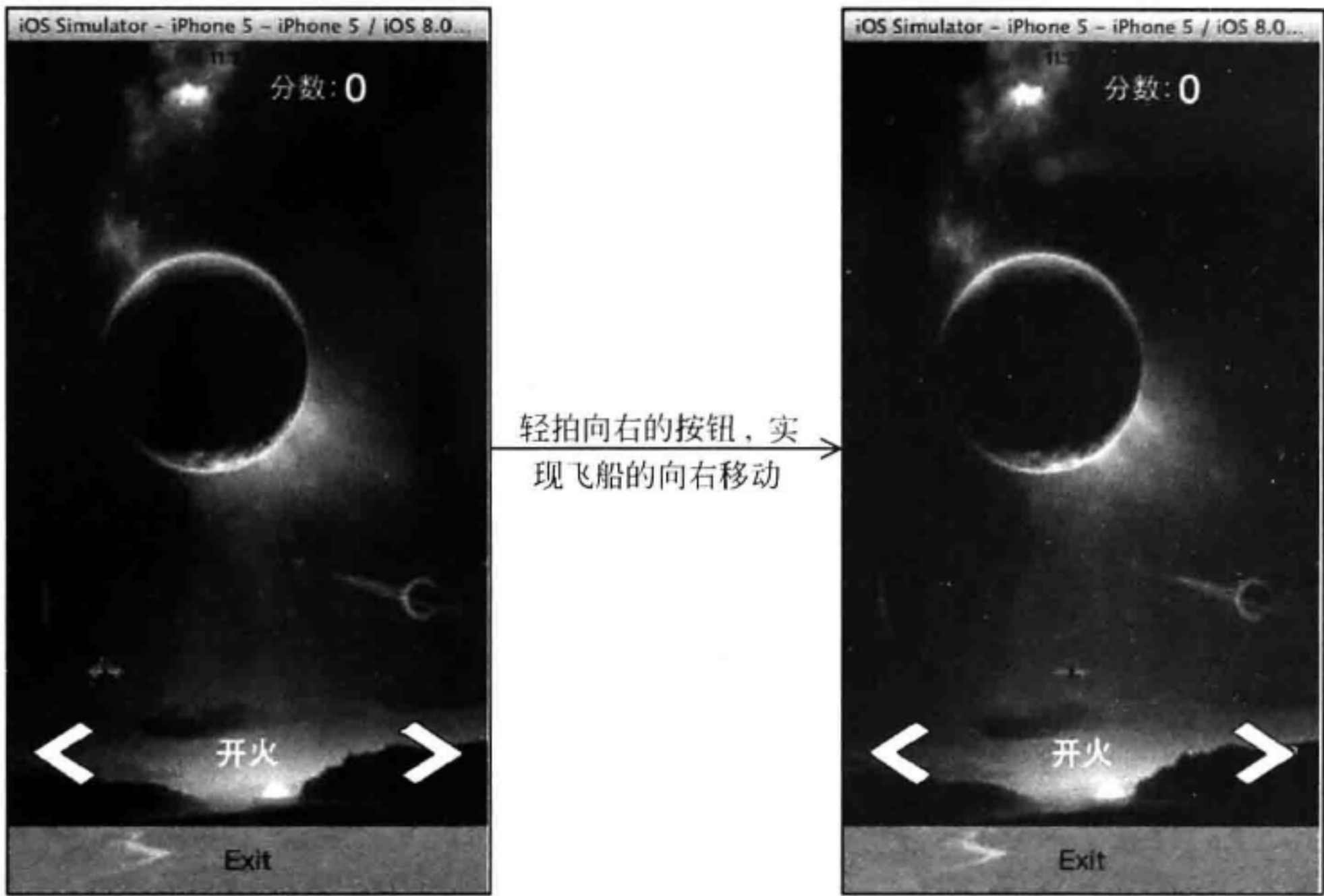


图 7.15 运行效果

当玩家轻拍向左的按钮后，飞船会向左移动，如图 7.16 所示。

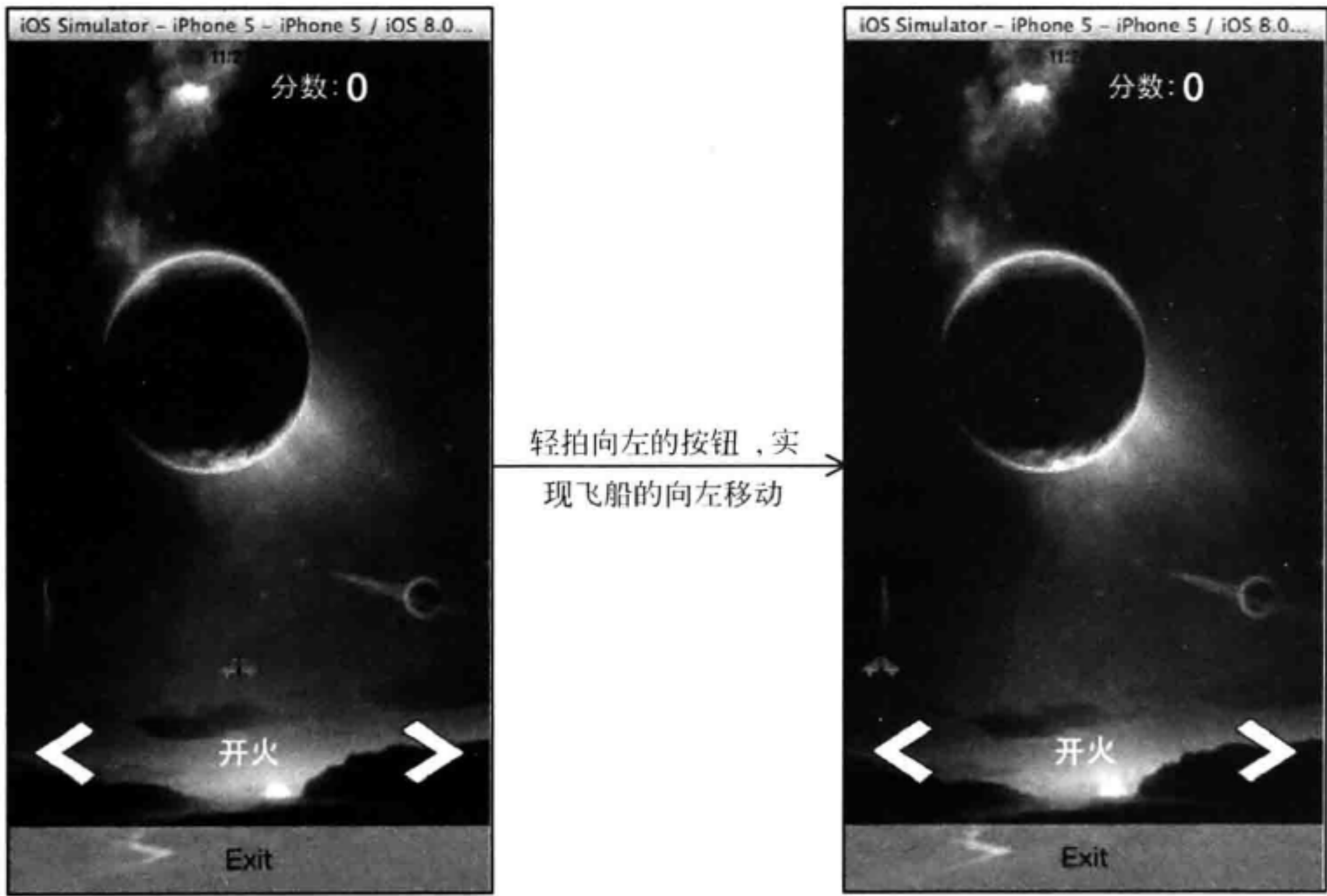


图 7.16 运行结果

3. 发射子弹

在上一章中，我们为飞船添加了发射子弹的功能。在本章中也不例外，对于子弹的创建和移动都和上一章中介绍的一样。首先，需要创建一个 Swift File 模板类型的文件，命名

为 PlayerBullet。然后在此文件中创建一个基于 NSObject 类的子类 PlayerBullet，该类用来编写关于子弹的一些设置，例如子弹的创建、动画效果等。代码如下：

```
import Foundation
import UIKit
class PlayerBullet:NSObject {
    var bulletRect:CGRect=CGRect()
    var bulletView:UIImageView=UIImageView()
    var bulletTimer:NSTimer?=nil
    var gameView:UIView=UIView()
    //发射子弹
    func fireBullet(gameView:UIView,playerView:UIImageView){
        self.gameView=gameView
        var bombImage1=UIImage(named: "bullet.png")
        var bulletStartX=playerView.frame.origin.x + (playerView.frame.size.
            width/2) - 4
        //截图
        var imageRef=CGImageCreateWithImageInRect(bombImage1.CGImage, CGRect
            Make (0, 0, 32, 64))
        var imageRef2=CGImageCreateWithImageInRect(bombImage1.CGImage,CGRect
            Make (33, 0, 32, 64))
        var bombArray:NSMutableArray=NSMutableArray() //创建一个可变数组对象
        //为数组添加对象
        bombArray.addObject(UIImage(CGImage: imageRef))
        bombArray.addObject(UIImage(CGImage: imageRef2))
        self.bulletRect=CGRectMake(bulletStartX, playerView.frame.origin.
            y, 8, 16)
        // bulletView 图像视图对象用来显示子弹
        self.bulletView=UIImageView(frame: bulletRect)//初始化图像视图的框架
        self.bulletView.image=UIImage(CGImage: imageRef2)
        //实现子弹图像切换的动画效果
        self.bulletView.animationImages=bombArray
        self.bulletView.animationDuration=0.3
        self.gameView.addSubview(self.bulletView)
        self.bulletView.startAnimating()
        //创建定时器
        self.bulletTimer=NSTimer.scheduledTimerWithTimeInterval(0.03,
            target: self, selector: Selector("moveBullet"),userInfo:nil,repeats:
            true)
    }
    //向上移动子弹
    func moveBullet(){
        self.bulletRect=CGRectOffset(self.bulletRect, 0, -5)
        self.bulletView.frame=self.bulletRect
        //判断子弹的 y 的位置是否小于 10
        if(self.bulletRect.origin.y<10){
            self.remove()
        }
    }
    //移除移动并且使定时器失效
    func remove(){
        self.bulletRect=CGRectMake(0, 350, 32, 64)
        self.bulletView.removeFromSuperview() //移除 bulletView 图像视图对象
        self.bulletTimer?.invalidate()
        self.bulletTimer=nil
    }
}
```

当在 PlayerBullet.swift 文件中对子弹设置好之后就可以在轻拍“开火”按钮，实现飞

船中子弹的发射了。以下就是具体的实现方式，首先打开 `PlayerObject.swift` 文件，在此文件中实例化一个 `playerBullet` 的对象。代码如下：

```
var playerBullet=PlayerBullet()
```

添加一个 `fireBullet()` 的方法，实现子弹的发射功能。代码如下：

```
func fireBullet(){
    playerBullet.fireBullet(self.gameView, playerView: self.playerView)
}
```

最后，打开 `Game.swift` 文件，将 `Game.swift` 文件和射击游戏界面中的“开火”按钮进行动作 `fireButton` 的关联。当玩家轻拍此按钮后，就可以触发此动作。关联好动作后，就可以实现由玩家控制子弹的发射了，代码如下：

```
@IBAction func fireButton(sender: AnyObject) {
    //判断当前的状态是否为 PLAYING
    if(self.currentState != .PLAYING){
        return
    }
    self.playerOne.fireBullet()
}
```

此时运行程序，会看到如图 7.17 所示的效果。

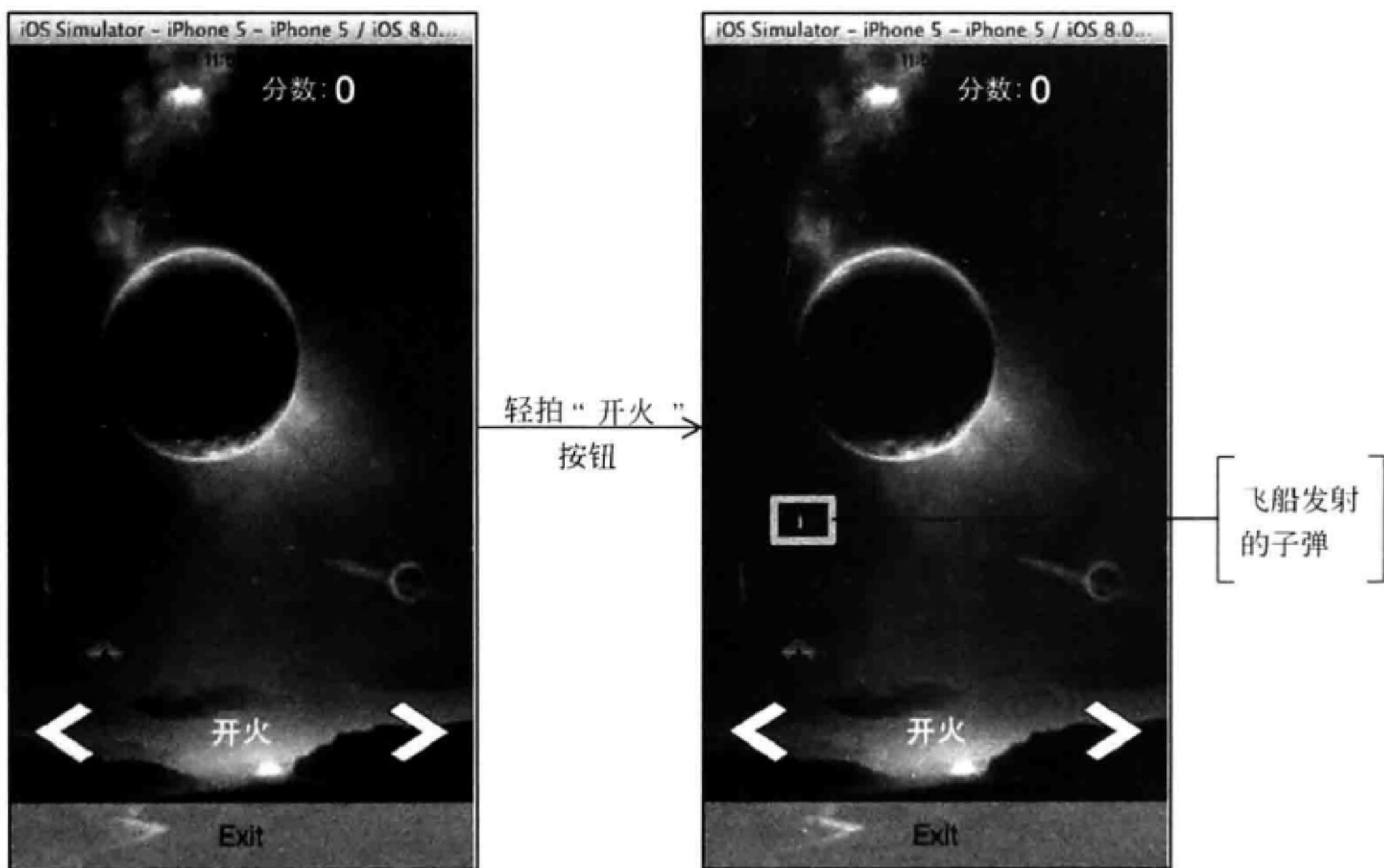


图 7.17 运行效果

7.6.3 敌人

在本章中的太空侵略者游戏中，敌人还是使用小蜜蜂来代替。本节将讲解对敌人的一些操作。

1. 创建敌人

在编写对敌人操作的代码时,首先,需要创建一个 Swift File 模板类型的文件,命名为 Enemy。然后在此文件中创建一个基于 NSObject 类的子类 Enemy,该类用来编写敌人的一些操作。首先就是创建敌人,在本章中创建敌人的方式和在上一章中介绍的方式是相同的,都是使用代码来创建。代码如下:

```
import Foundation
import UIKit
class Enemy: NSObject {
    var gameView:UIView=UIView()
    var minXPos:NSInteger=0
    var maxXPos:NSInteger=0
    var enemyList:NSMutableArray=NSMutableArray()
    var eSize:Int=32
    //用来设置敌人的行和列
    var enemyRows:Int=5
    var enemyColumns:Int=5
    var enemyView:UIImageView?=nil
    func initEnemies(gameView:UIView){
        self.minXPos=10
        self.maxXPos=278
        self.gameView=gameView
        var rowCount:Int=0
        var startX:Int=10
        var startY:Int=0
        var enemyImage=UIImage(named: "enemy01.png")
        var i:Int=0
        //创建敌人
        for(i;i<(enemyRows * enemyColumns);i++){
            var columnMod:Int=i%enemyColumns
            if(columnMod==0){
                rowCount++
            }
            //设置敌人的位置
            var xPos:Int=startX+((eSize*columnMod)+(columnMod*5))
            var yPos:Int=startY+((eSize*rowCount)+(rowCount*10))
            enemyView=UIImageView(image: enemyImage) //设置敌人显示的图像
            enemyView?.frame=CGRectMake(CGFloat(xPos),CGFloat(yPos),
            CGFloat(eSize), CGFloat(eSize)); //设置敌人的框架
            enemyView?.tag=i+1
            self.enemyList.addObject(enemyView!) //为数组添加对象
            self.gameView.addSubview(enemyView!) //添加敌人
        }
    }
}
```

创建好敌人后, Game.swift 文件将创建的敌人显示在射击游戏界面中。首先需要实例化一个关于敌人的对象,代码如下:

```
var enemies:Enemy=Enemy()
```

然后,需要在 loadingScreen()方法中调用 Enemy 类中的 initEnemies(gameView:UIView)方法,实现敌人的显示。代码如下:

```
self.playerView=self.playerOne.getPlayerView()
self.enemies.initEnemies(self.view)
```


此时运行程序，可以看到如图 7.18 所示的效果。

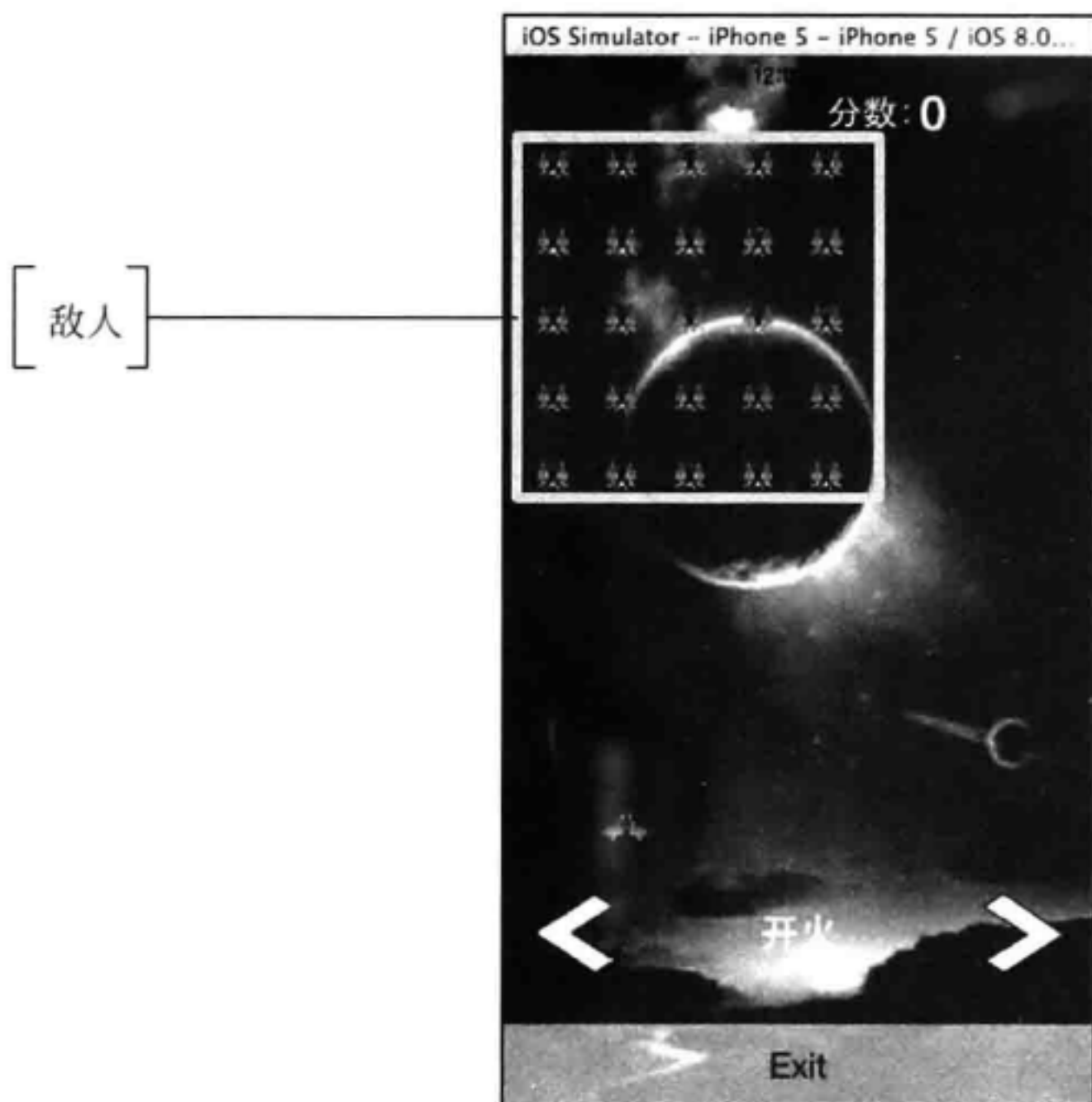


图 7.18 敌人

2. 移动敌人

敌人的移动也和上一章中介绍的方式是一样的，需要使用定时器进行实现。首先需要声明两个变量，代码如下：

```
var enemyTimer:NSTimer?=nil
var goingLeft :Bool=false
```

然后添加一个 `startTimers()` 方法，在此方法中设置定时器，其代码如下：

```
func startTimers(){
    self.enemyTimer=NSTimer.scheduledTimerWithTimeInterval(0.03, target:
self, selector: Selector("moveEnemies"), userInfo: nil, repeats: true)
}
```

定时器设置好后，就可以实现敌人的移动了，其代码如下：

```
func moveEnemies(){
    var enemyView:UIImageView=self.enemyList[0] as UIImageView
    //判断敌人的 x 的位置是否小于最小的 x 的位置，即 10
    if(Int(enemyView.frame.origin.x) <= self.minXPos){
        goingLeft = false
    }
    enemyView = self.enemyList[enemyColumns-1] as UIImageView
    //判断敌人的 x 的位置是否大于最小的 x 的位置，即 278
    if(Int(enemyView.frame.origin.x)>=self.maxXPos){
        goingLeft = true
    }
    var i:Int=0
    //遍历数组
    for(i;i<self.enemyList.count; i++){
```



```

enemyView = self.enemyList[i] as UIImageView;
var xPos:Int=0
//判断 goingLeft 的值是否为 true
if(goingLeft){
    xPos = Int(enemyView.frame.origin.x)-3;
}else{
    xPos = Int(enemyView.frame.origin.x)+3;
}
enemyView.frame = CGRectMake(CGFloat(xPos), enemyView.frame.origin.y,
CGFloat(eSize), CGFloat(eSize)); //设置敌人的框架
self.gameView.addSubview(enemyView)
}
}

```

最后在 Game.swift 文件中调用 Enemy 文件中的 startTimers() 方法。代码如下：

```

func closeScreen() {
    self.loadingView.removeFromSuperview()
    self.enemies.startTimers()
}

```

此时运行程序，可以看到如图 7.19 所示的效果。

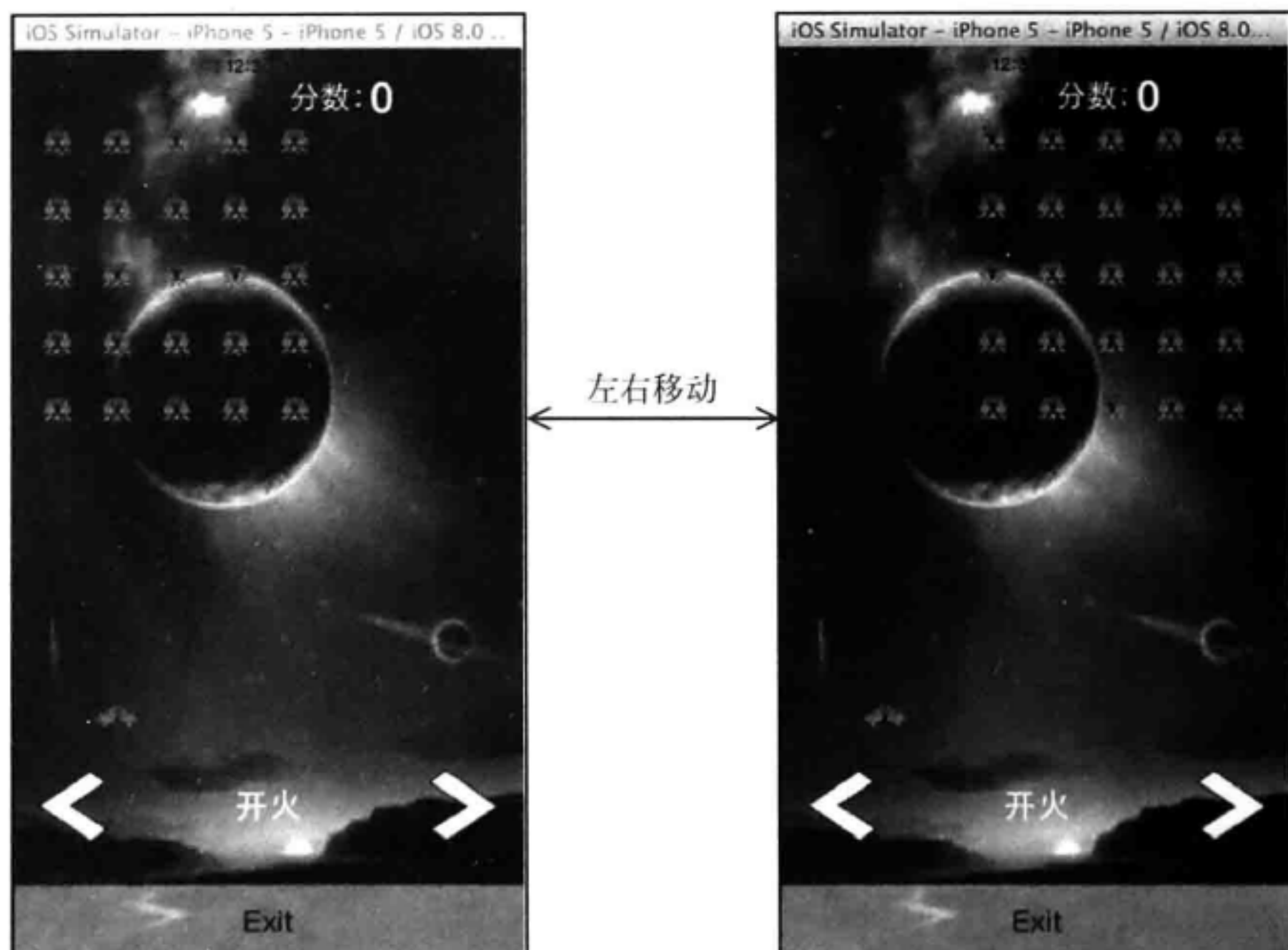


图 7.19 运行效果

3. 发射子弹

子弹的发射同样也是和上一章中所讲的方式相同。首先需要创建一个 Swift File 模板类型的文件，命名为 EnemyBullet。然后在此文件中创建一个基于 NSObject 类的子类 EnemyBullet，该类是用来编写对敌人的子弹的一些设置，如创建子弹等。代码如下：

```

import Foundation
import UIKit
class EnemyBullet: NSObject {
    var bombRect:CGRect=CGRect()
}

```

```

var bombView:UIImageView=UIImageView()
var bombTimer:NSTimer?=nil
var isActive:Bool=false
var gameView:UIView=UIView()
//实现子弹的发射
func fireBullet(gameView:UIView,enemyList:NSArray) {
    self.isActive=true
    var randEnemy:Int=random()%enemyList.count
    var enemyView:AnyObject=enemyList[randEnemy]
    self.gameView=gameView
    var bombImage1:UIImage=UIImage(named: "bullet.png")
    var bombStartX=enemyView.frame.origin.x
    //截图
    var imageRef=CGImageCreateWithImageInRect(bombImage1.CGImage, CGRectMake(0, 0, 32, 64))
    var imageRef2=CGImageCreateWithImageInRect(bombImage1.CGImage, CGRectMake(33, 0, 32, 64))
    var bombArray:NSMutableArray=NSMutableArray()
    bombArray.addObject(UIImage(CGImage: imageRef))
    bombArray.addObject(UIImage(CGImage: imageRef2))
    self.bombRect=CGRectMake(bombStartX, enemyView.frame.origin.y, 8, 16)
    self.bombView.frame=self.bombRect
    self.bombView.image=UIImage(CGImage: imageRef2)
    //实现图像的切换动画
    self.bombView.animationImages=bombArray
    self.bombView.animationDuration=3
    self.gameView.addSubview(self.bombView)
    self.bombView.startAnimating()
    self.bombTimer=NSTimer.scheduledTimerWithTimeInterval(0.03,
    target: self, selector: Selector("moveBomb"), userInfo: nil,
    repeats:true) //实现对定时器的设置
}
//向下移动子弹
func moveBomb() {
    self.bombRect=CGRectOffset(self.bombRect, 0, 5)
    self.bombView.frame=self.bombRect
    //判断子弹的 y 的位置是否大于 440
    if(self.bombRect.origin.y>440) {
        self.isActive=false
        self.bombTimer?.invalidate()
        self.bombTimer=nil
        self.bombView.removeFromSuperview()
    }
}
}

```

对敌人的子弹设置好，就可以实现敌人发射子弹的功能了。敌人发射子弹不是由玩家操作的，所以需要使用定时器，让子弹每隔一段时间发射一颗子弹。具体的实现方式介绍如下。首先打开 Enemy.swift 文件，在此文件中创建两个变量，代码如下：

```

var enemyBulletTimer:NSTimer?=nil
var enemiesBullet:EnemyBullet?=EnemyBullet()

```

然后在此文件的 startTimers()方法中添加对定时器对象 enemyBulletTimer 的设置。代码如下：

```

self.enemyTimer=NSTimer.scheduledTimerWithTimeInterval(0.03, target: self,
selector: Selector("moveEnemies"), userInfo: nil, repeats: true)
self.enemyBulletTimer=NSTimer.scheduledTimerWithTimeInterval(1, target:

```



```
self, selector: Selector("dropBomb"), userInfo: nil, repeats: true)
```

最后实现子弹的发射效果。代码如下：

```
func dropBomb() {
    if (self.enemiesBullet == nil || self.enemiesBullet?.isActive == false) {
        enemiesBullet!.fireBullet(self.gameView, enemyList: self.enemyList)
    }
}
```

此时运行程序，可以看到如图 7.20 所示的效果。

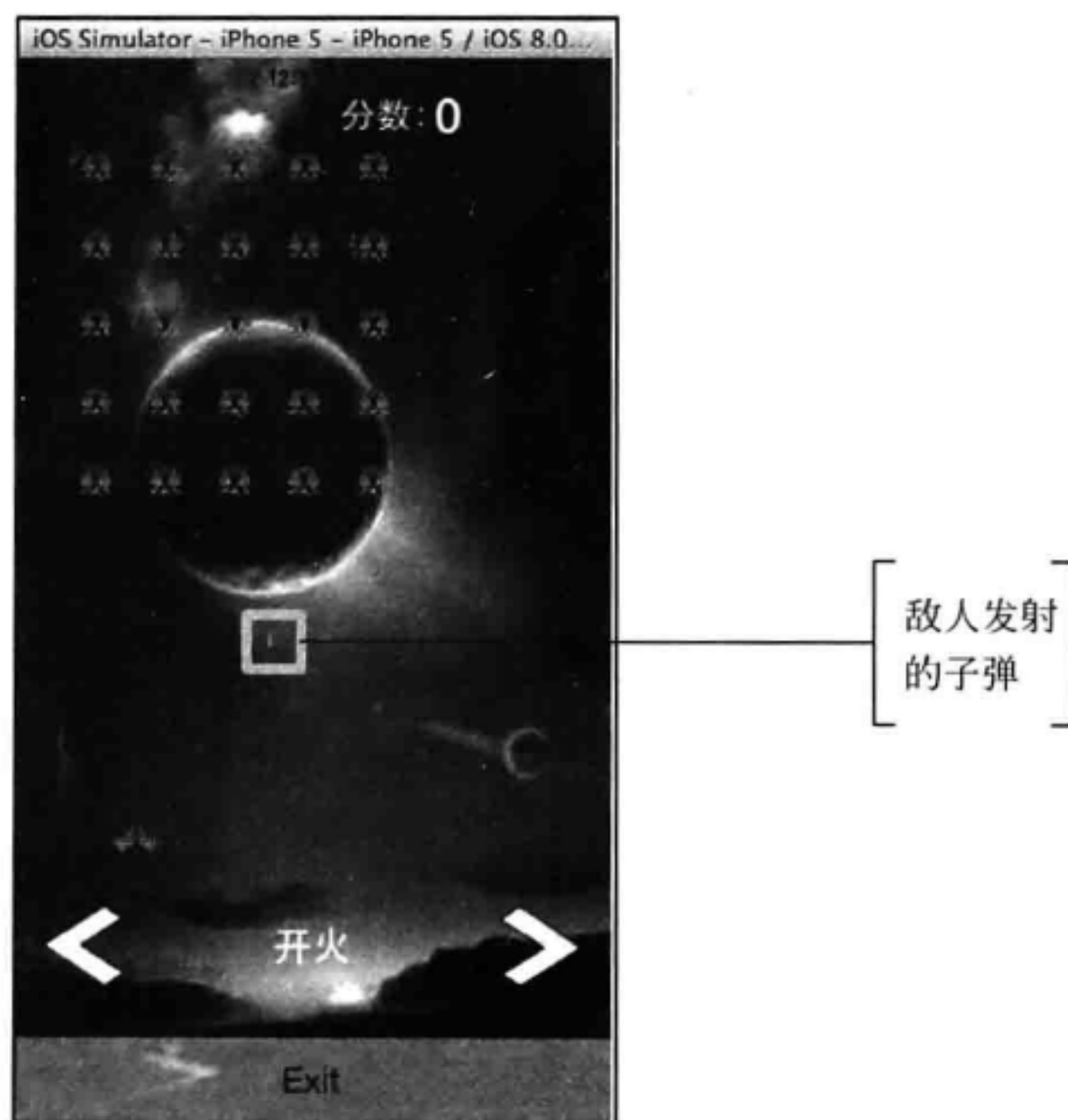


图 7.20 运行效果

7.7 检测碰撞

以上这些内容大部分和上一章内容一样。在本节中将讲解如何检测两个物体的碰撞，由此来实现敌人的子弹击中飞船和飞船的子弹击中敌人的功能。

7.7.1 敌人的子弹击中飞船的检测

首先，我们来看敌人的子弹击中飞船。对于这一功能，我们可以使用 `CGRectIntersectsRect(_rect1:CGRect, _rect2:CGRect)` 方法来实现。此方法实现的功能是判断矩形结构是否交叉或者两个矩形对象是否重叠。其语法形式如下：

```
CGRectIntersectsRect(_rect1: CGRect, _rect2: CGRect);
```

其中，`_rect1` 用来指定第一个矩形对象，`_rect2` 用来指定第二个矩形对象。该方法的返回值类型为布尔类型。当 Bool 为 true 时，表示两个矩形对象重叠；当 Bool 为 false 时，

表示两个矩形对象没有重叠。判断敌人的子弹是否击中飞船，就是要对敌人的子弹的矩形和飞船的矩形进行判断，看一下这两个矩形是否重叠。如果重叠表明敌人的子弹击中了飞船，如果没有重叠表明敌人的子弹没有击中飞船。在 Game.swift 文件中编写以下代码：


```
func intersectCheck() {
    var isConnecting:Bool=CGRectIntersectsRect(self.enemies.enemiesBullet!.
    bombRect, self.playerOne.playerRect);
    //判断敌人的子弹是否击中飞船
    if(isConnecting==true){
        NSTimer.scheduledTimerWithTimeInterval(2, target: self, selector:
        Selector("endScreen"), userInfo: nil, repeats: false)
        self.changeState(.RELOADING)
    }
}
```

当敌人的子弹击中飞船后，就会结束游戏。首先实例化一个关于 UIViewController 的实例变量，代码如下：

```
var back:UIViewController=UIViewController()
```

然后在 endScreen()方法中添加以下的代码：

```
func endScreen() {
    self.loadingView.removeFromSuperview()
    self.changeState(.ENDING)
    //退回到主菜单
    back=self.storyboard!.instantiateViewControllerWithIdentifier("back")
    as UIViewController
    self.view.addSubview(back.view)
}
```

 **注意：**在 instantiateViewControllerWithIdentifier()方法中的 back 就是在 Storyboard ID 中输入的，在 7.3 节中进行过设置。

此时运行程序，会看到如图 7.21 所示的效果。

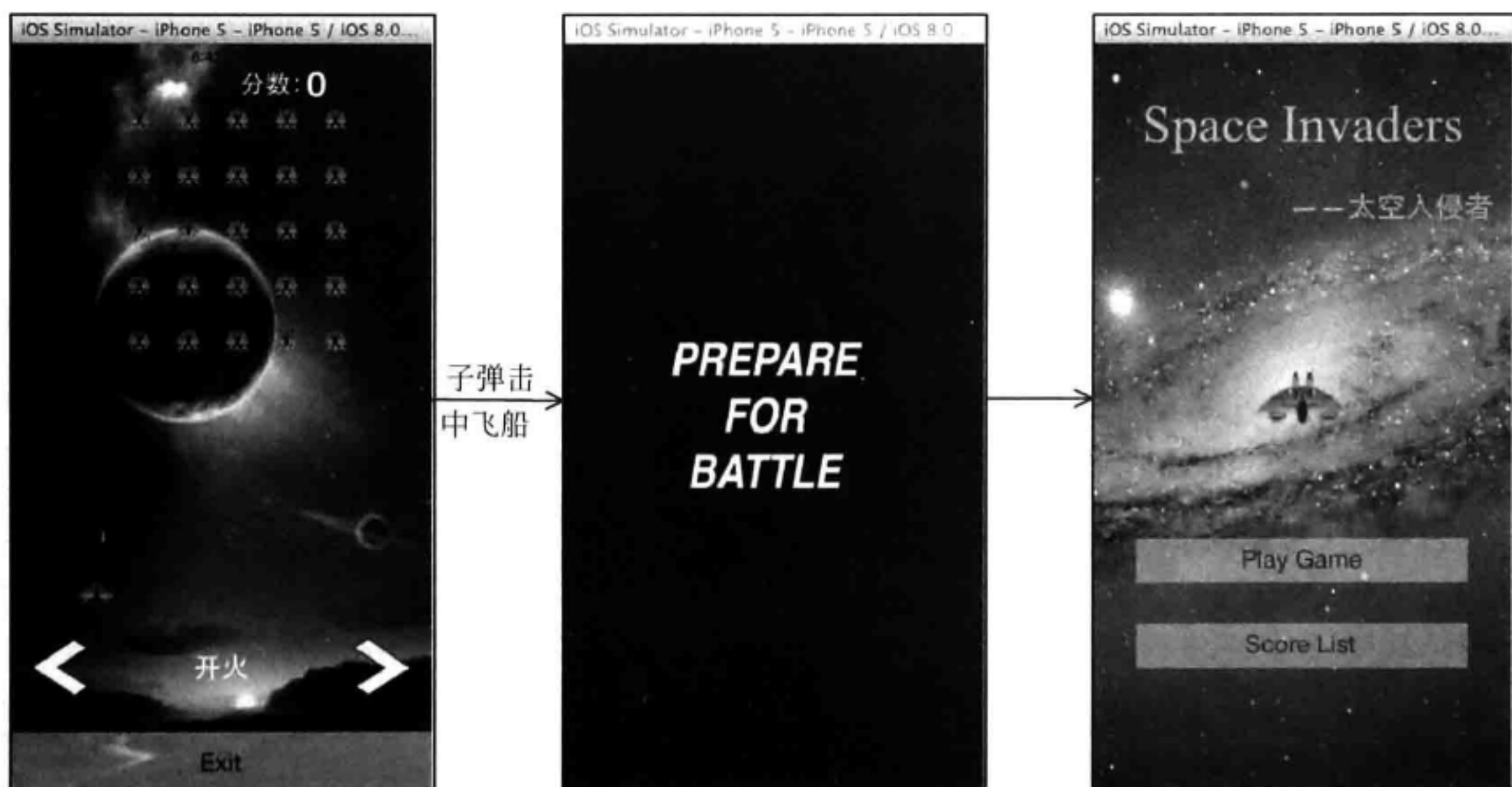


图 7.21 运行效果

7.7.2 飞船的子弹击中敌人的检测

敌人的子弹可以击中飞船，相反，由飞船发射的子弹也可以击中敌人。它也是使用 `CGRectIntersectsRect(_rect1:CGRect, _rect2:CGRect)` 方法实现的。以飞船的子弹击中 `tag` 值为 25 的敌人来说，首先，在 `intersectCheck()` 方法中添加一个 `isConnecting24` 的变量，此变量用来存储布尔值。此布尔值是飞船子弹的矩形对象是否击中 `tag` 为 25 的敌人的值。代码如下：

```
var isConnecting24:Bool=CGRectIntersectsRect(self.enemies.enemyList[24].
frame, self.playerOne.playerBullet.bulletRect);
```

然后在 `intersectCheck()` 方法中实现如果击中 `tag` 为 25 的敌人后执行的方法。代码如下：

```
if(isConnecting24==true){
    NSTimer.scheduledTimerWithTimeInterval(0.0002, target: self, selector:
        Selector("hidden24"), userInfo: nil, repeats: false)
}
```

在上面的代码中会看到，在创建的定时器中，定时器每隔 0.0002 就会调用 `hidden24()` 方法。此方法实现的功能是隐藏 `tag` 为 25 的敌人，即隐藏 `tag` 为 25 的图像视图对象，并且将击中敌人的子弹移除，代码如下：

```
func hidden24(){
    var image=self.enemies.gameView.viewWithTag(25)
    //判断飞船是否隐藏
    if(image?.hidden==false){
        image?.hidden=true
        self.playerOne.playerBullet.remove()
    }
}
```

此时运行程序，会看到如图 7.22 所示的效果。

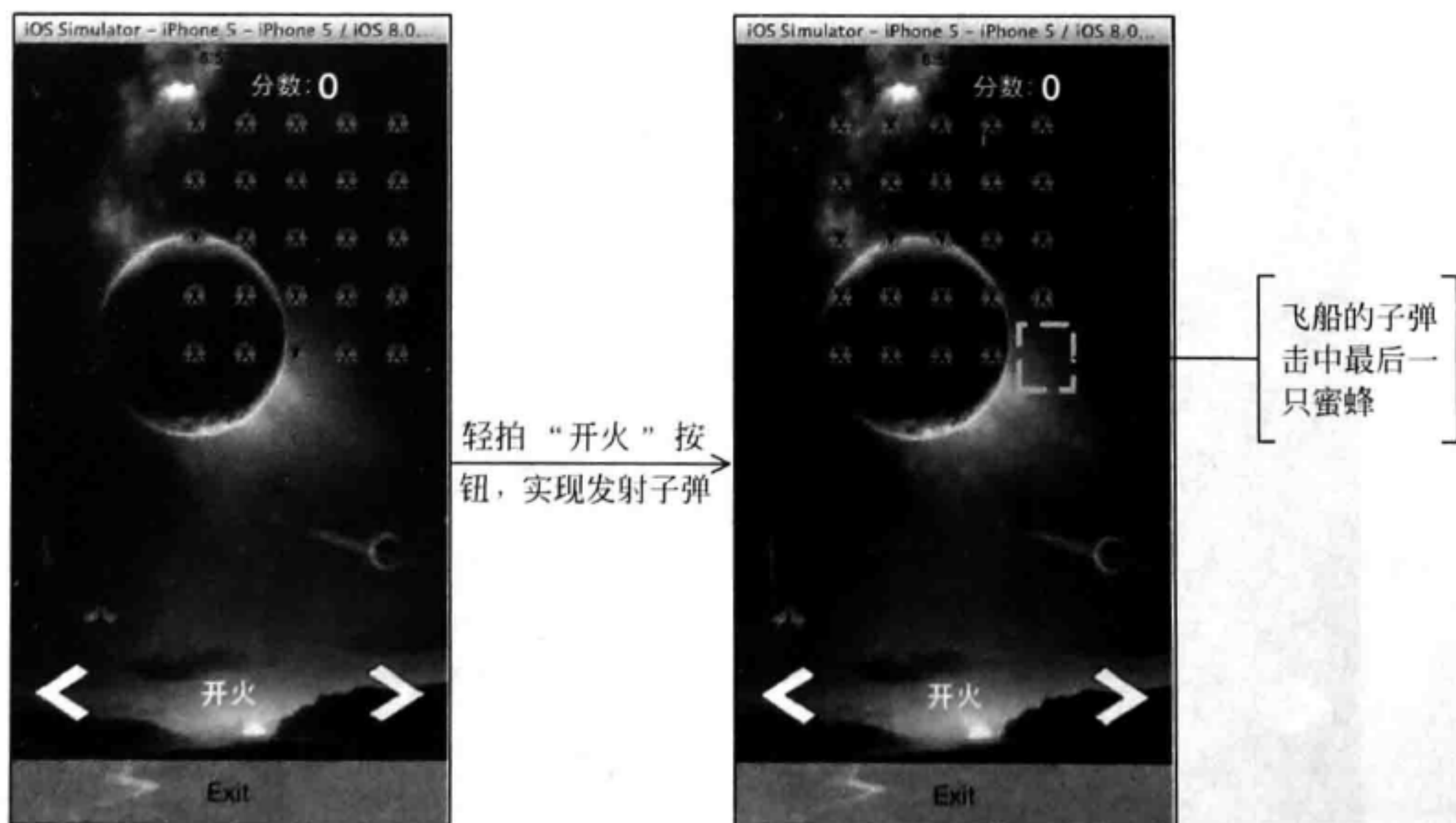


图 7.22 运行效果

⚠注意：以上的方法只是子弹击中 tag 值为 25 的敌人，使用类似的方法还可以实现子弹击中 tag 值为 1~24 的敌人。

7.8 计 分 功 能

在一个正规的太空侵略者的游戏中，可以看到在飞船每击中一个敌人后，都会有相应分数的显示。以下就来实现这一功能。首先在 Game.swift 文件中声明一个变量 j，它是用来计算分数的。代码如下：

```
var j:Int=0
```

然后在 hidden0()~hidden24()的方法中添加以下的代码：

```
if (image?.hidden==false) {
    image?.hidden=true
    self.playerOne.playerBullet.remove()
    //需要添加的代码
    j=j+10
}
```

如果要在标签中显示分数，首先需要将射击游戏界面的 label2 进行插座变量 scoreLabel 的声明和关联。声明代码如下：

```
@IBOutlet weak var scoreLabel: UILabel!
```

然后在 hidden0()~hidden24()的方法中添加以下代码：

```
j=j+10
scoreLabel.text="\ (j) "
```

此时运行程序，会看到如图 7.23 所示的效果。

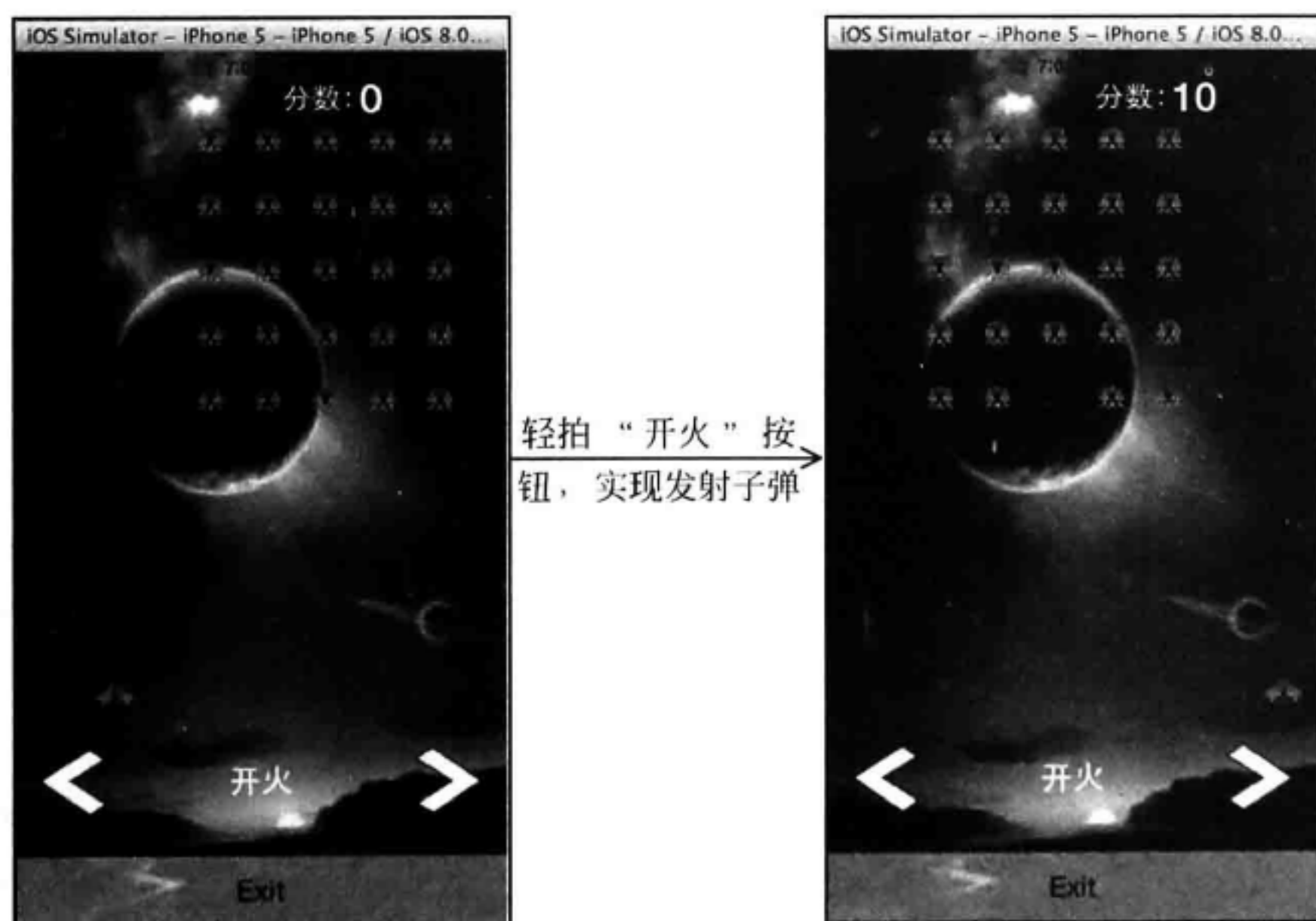


图 7.23 运行效果

7.9 歼灭所有敌人

一个游戏的结束不仅可以通过敌人的子弹击中飞船来实现，还可以通过消灭所有敌人来实现。以下就是通过飞船的子弹击中所有敌人来结束游戏的。首先需要添加一个新的 View Controller 视图控制器对象。然后打开 Show the Identity inspector，即标识查看器，在其中找到 Storyboard ID，在此输入框中输入 completevc，然后将 Use Storyboaed ID 单选框选中。最后，对此视图控制器的界面进行设计，效果如图 7.24 所示。

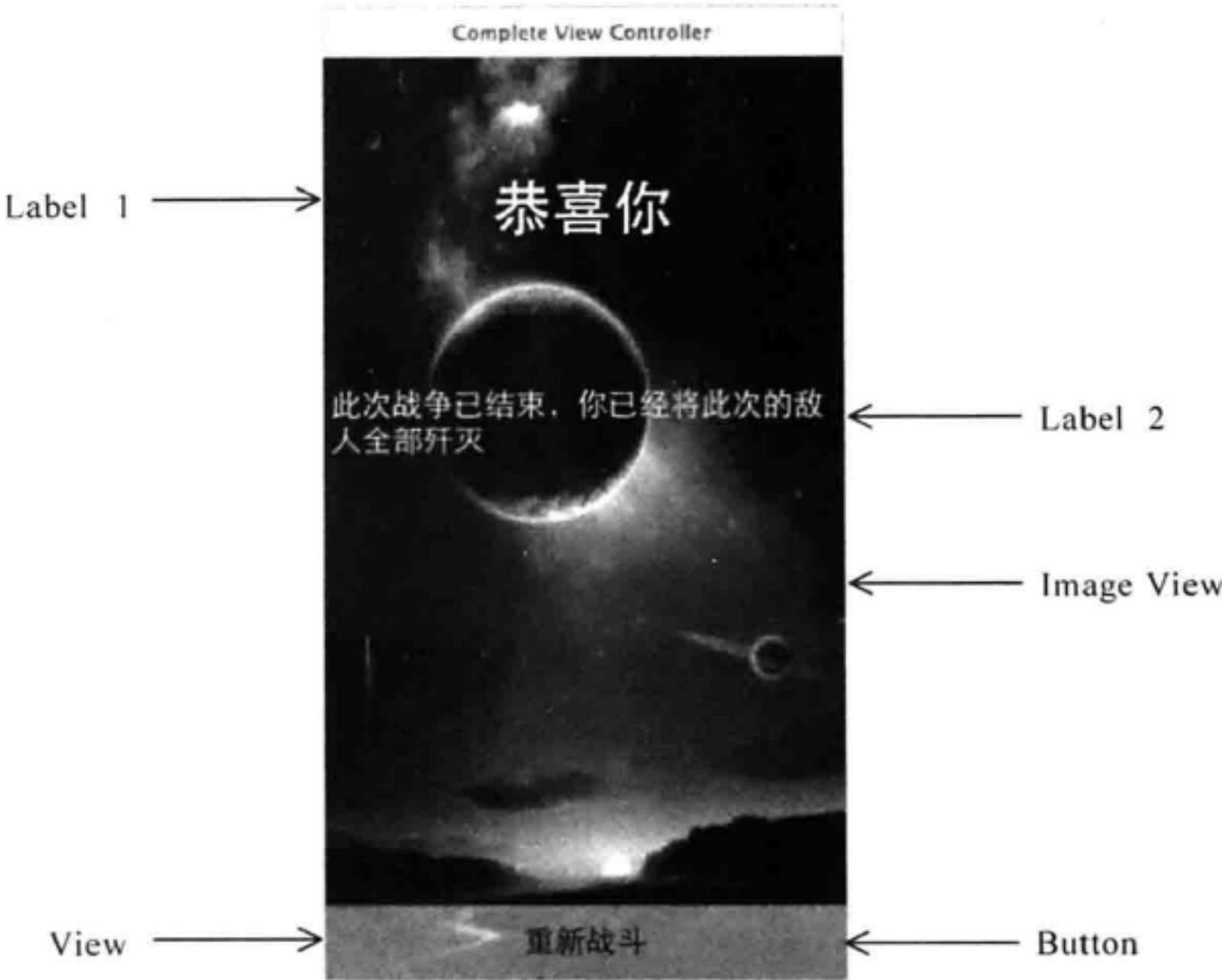


图 7.24 界面效果

需要添加的视图对象，以及对它们的设置，如表 7-3 所示。

表 7-3 设置界面

视 图	设 置
Label1	Text: 恭喜你 Color: 白色 Font: System Bold 37.0 Alignment: 居中 位置和大小: (28, 68, 264, 50)
Label2	Text: 此次战争已结束，你已经将此次的敌人全部歼灭 Color: 白色 Font: System 19.0 Lines: 2 位置和大小: (4, 191, 316, 67)
Image View	Image: backdrop.jpg 位置和大小: (0, 0, 320, 568)

续表

视 图	设 置
View	Alpha: 0.5 位置和大小: (0, 523, 320, 45)
Button	Title: 重新战斗 Font: Helvetica Neue 19.0 Text Color: 黑色 位置和大小: (114, 7, 93, 30)

界面设计好后，打开 Game.swift 文件，实例化一个对象 cvc，代码如下：

```
var cvc:UIViewController=UIViewController()
```

然后在 hidden0()~hidden24()的方法中添加一个定时器，代码如下：

```
j=j+10
scoreLabel.text="\ (j) "
NSTimer.scheduledTimerWithTimeInterval(0.0002, target: self, selector:
Selector("complete"), userInfo: nil, repeats: false)
```

最后，添加一个 complete()方法，在此方法中实现判断敌人是否全部被歼灭，以及跳转场景的功能。其中，判断敌人是否全部被歼灭可以使用 j 来实现，如果 j 为 250 就表明敌人全部歼灭，代码如下：

```
func complete() {
    //判断 j 是否为 250
    if (j==250) {
        self.changeState(.RELOADING)
        cvc=self.storyboard!.instantiateViewControllerWithIdentifier
        ("completevc") as UIViewController
        self.view.addSubview(cvc.view)
    }
}
```

此时运行程序，可以看到如图 7.25 所示的效果。

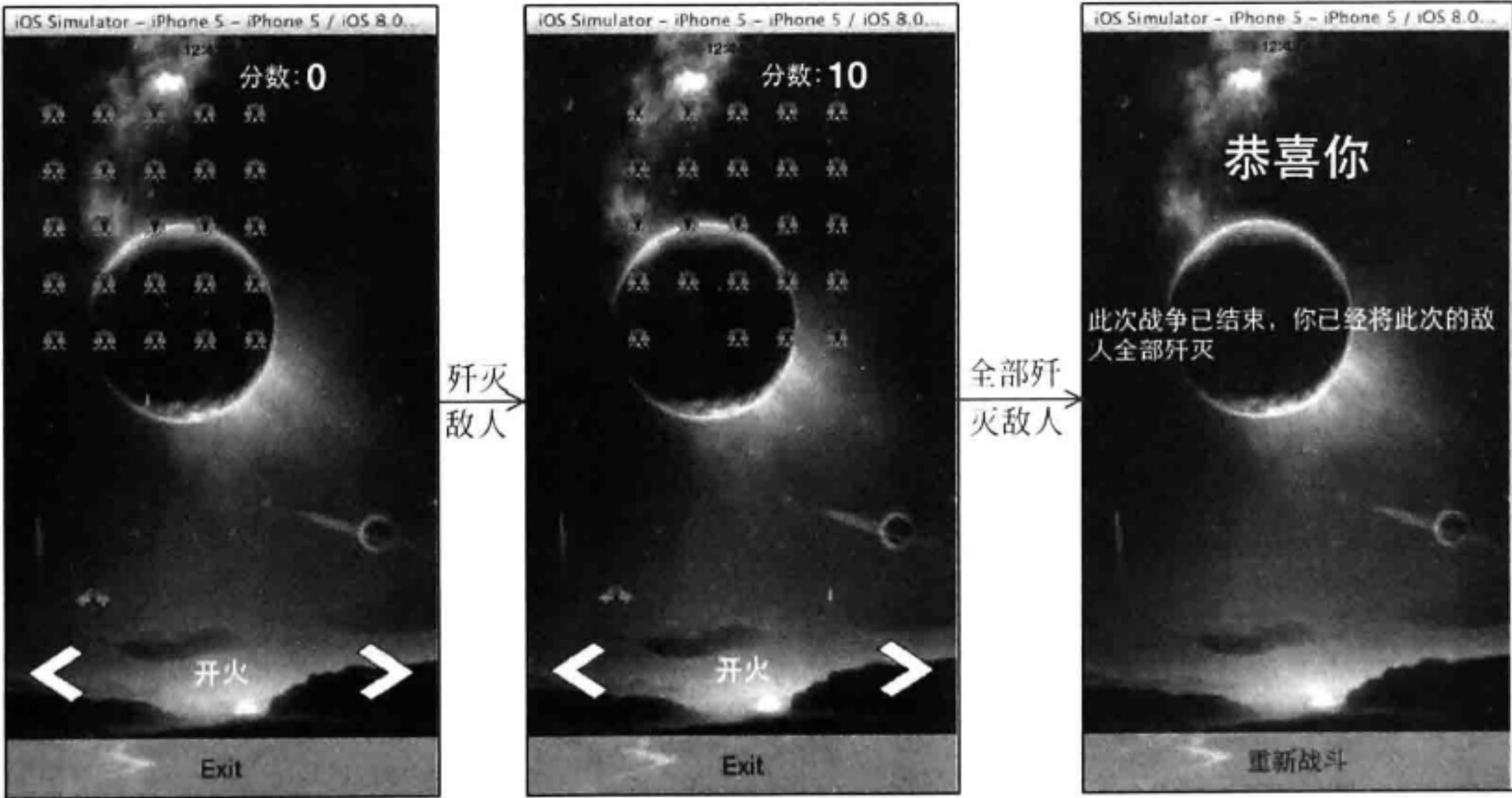


图 7.25 运行效果

7.10 分数榜模块

本节将讲解有关分数榜的一些操作，如界面设计和实现分数的显示等。

7.10.1 准备工作

在对界面进行设计前，首先需要创建一个 Swift File 模板类型的文件，命名为 **Score**。然后在此文件中创建一个基于 **UIViewController** 类的子类 **ListViewController**，该类用于编写显示分数的代码。

7.10.2 界面设计

在分数榜中显示关于分数的数据可以使用标签实现，也可以使用按钮实现等。本小节还是使用表视图对象来显示分数，以下就是对界面设计的具体步骤。

- (1) 在视图对象库中拖动 **View Controller** 视图控制器对象到画布中。
- (2) 单击新添加的视图控制器，选择界面上方的 **Dock** 中的 **View Controller** 图标。在工具窗口中的 **Show the Identity inspector** 选项，即标示查看器，将 **Custom Class** 下的 **Class** 设置为创建的 **ListViewController** 类。这时，在画布中的这个视图控制器就变为了 **List View Controller** 视图控制器。
- (3) 对 **List View Controller** 视图控制器的界面进行设计，效果如图 7.26 所示。

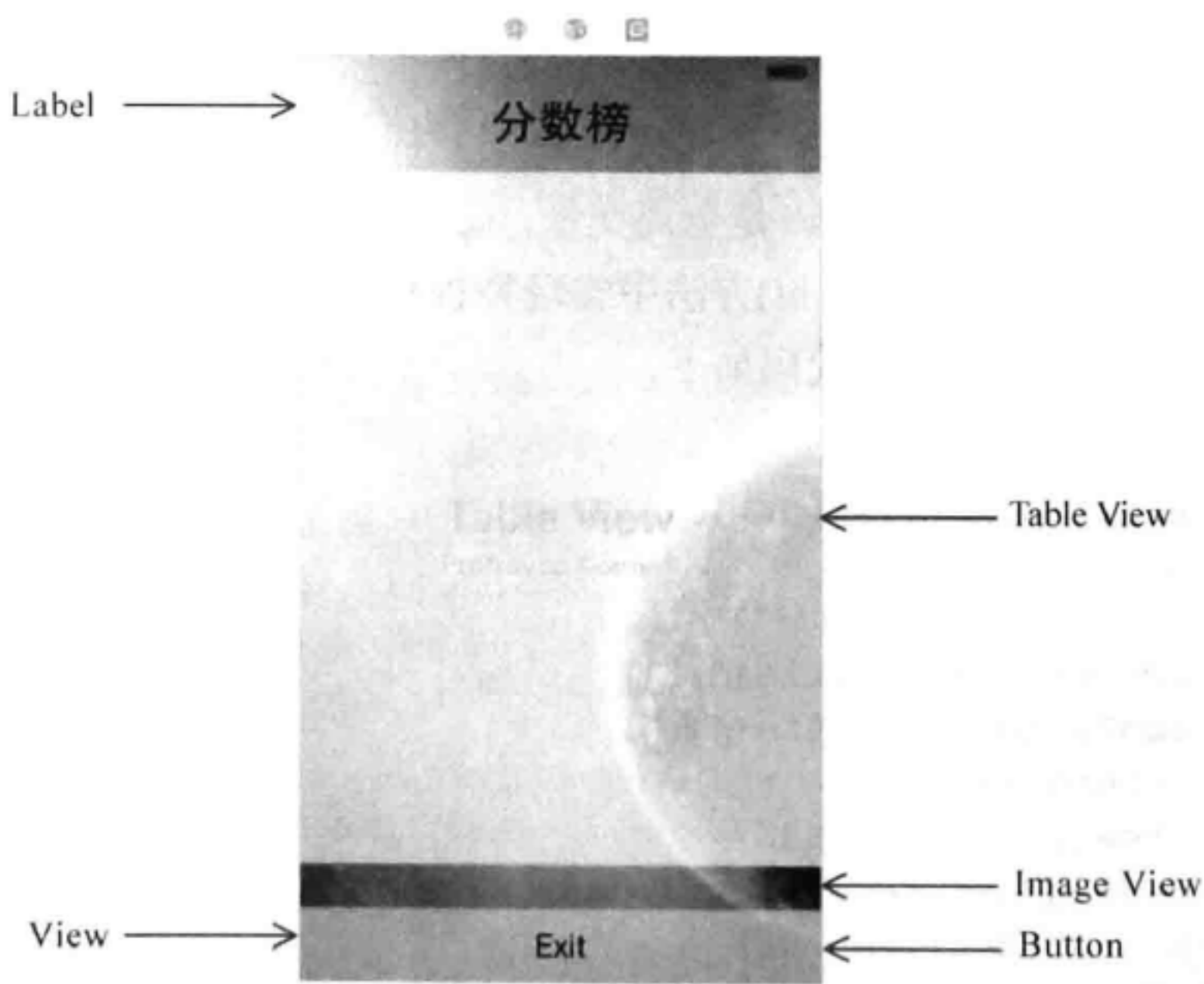


图 7.26 界面的效果

需要添加的视图对象，以及对它们的设置，如表 7-4 所示。

表 7-4 设置界面

视 图	设 置
Image View	Image: background1.jpg 位置和大小: (0, 0, 320, 568)
Label	Text: 分数榜 Font: System Bold 29.0 Alignment: 居中 位置和大小: (81, 20, 158, 40)
Table View	Background: 透明 与dataSource关联 与delegate关联 位置和大小: (0, 72, 320, 424)
View	Alpha: 0.5 Background: 白色 位置和大小: (0, 523, 320, 45)
Button	Title: Exit Font: Helvetica Neue 19.0 Text: 白色 位置和大小: (137, 8, 46, 30)

7.10.3 实现分数的显示

以下是在表视图中显示分数的具体步骤。
首先，需要在 Game.swift 文件中实例化一些对象。代码如下：

```
var score:NSMutableArray=NSMutableArray()  
var time:NSMutableArray=NSMutableArray()  
var back:UIViewController=UIViewController()
```

在上面的代码中，score 和 time 是全局变量，score 用来保存当前的分数，time 用来保存当前的时间。其次，在 endScreen()方法中实现警告视图的添加以及设置，此警告视图用来实现对数据是否保存的操作。代码如下：

```
func endScreen() {  
    var alert:UIAlertView=UIAlertView()  
    alert.title="游戏结束"  
    alert.message="是否保存分数"  
    alert.addButtonWithTitle("是")  
    alert.addButtonWithTitle("否")  
    alert.delegate=self  
    alert.show()  
    self.loadingView.removeFromSuperview()  
    self.changeState(.ENDING)  
    back=self.storyboard!.instantiateViewControllerWithIdentifier("back")  
    as UIViewController  
    self.view.addSubview(back.view)  
}
```

然后实现警告视图按钮的响应，代码如下：

```
func alertView(_alertView: UIAlertView,
  clickedButtonAtIndex buttonIndex: Int){
    var name:NSString=_alertView.buttonTitleAtIndex(buttonIndex)
    //判断当前轻拍的按钮是否为“是”
    if(name.isEqualToString("是")){
        score.addObject(j)                                //保存分数
        //日期和字符串的转换
        var dateFormatter:NSDateFormatter=NSDateFormatter()
        dateFormatter.dateFormat="yy//MM//dd HH:mm:ss"
        var d:NSDate=NSDate()                                //获取当前日期
        var date:NSString=dateFormatter.stringFromDate(d)
        time.addObject(date)                                //保存时间
    }
}
```

最后，打开 Score.swift 文件，编写代码，实现在表视图中显示分数的功能。代码如下：

```
import UIKit
class ListViewController: UIViewController {
    //视图在加载后调用
    override func viewDidLoad() {
        super.viewDidLoad()
    }
    //返回节数
    func numberOfSectionsInTableView(tableView: UITableView)->Int{
        return 1
    }
    //返回行数
    func tableView(tableView: UITableView,numberOfRowsInSection section:
    Int) -> Int{
        return score.count
    }
    //返回表视图的表单元格
    func tableView(tableView: UITableView,cellForRowAtIndexPath indexPath:
    NSIndexPath) ->UITableViewCell{
        var CellIdentifier:NSString="Cell"
        var cell: UITableViewCell? = tableView.dequeueReusableCellWithIdentifier
        (CellIdentifier) as? UITableViewCell
        //判断表单元格是否为空
        if(cell == nil){
            cell=UITableViewCell(style: UITableViewCellStyle.Value1,
            reuseIdentifier: CellIdentifier)
            cell?.backgroundColor=UIColor.clearColor()
            cell?.detailTextLabel?.textColor=UIColor.blackColor()
        }
        cell!.textLabel?.text="\ (score.objectAtIndex(indexPath.row)) "
        cell!.detailTextLabel?.text="\ (time.objectAtIndex(indexPath.row)) "
        return cell!
    }
}
```

此时运行程序，会看到如图 7.27 所示的效果。当轻拍“开火”按钮时，可以实现对敌人的攻击。

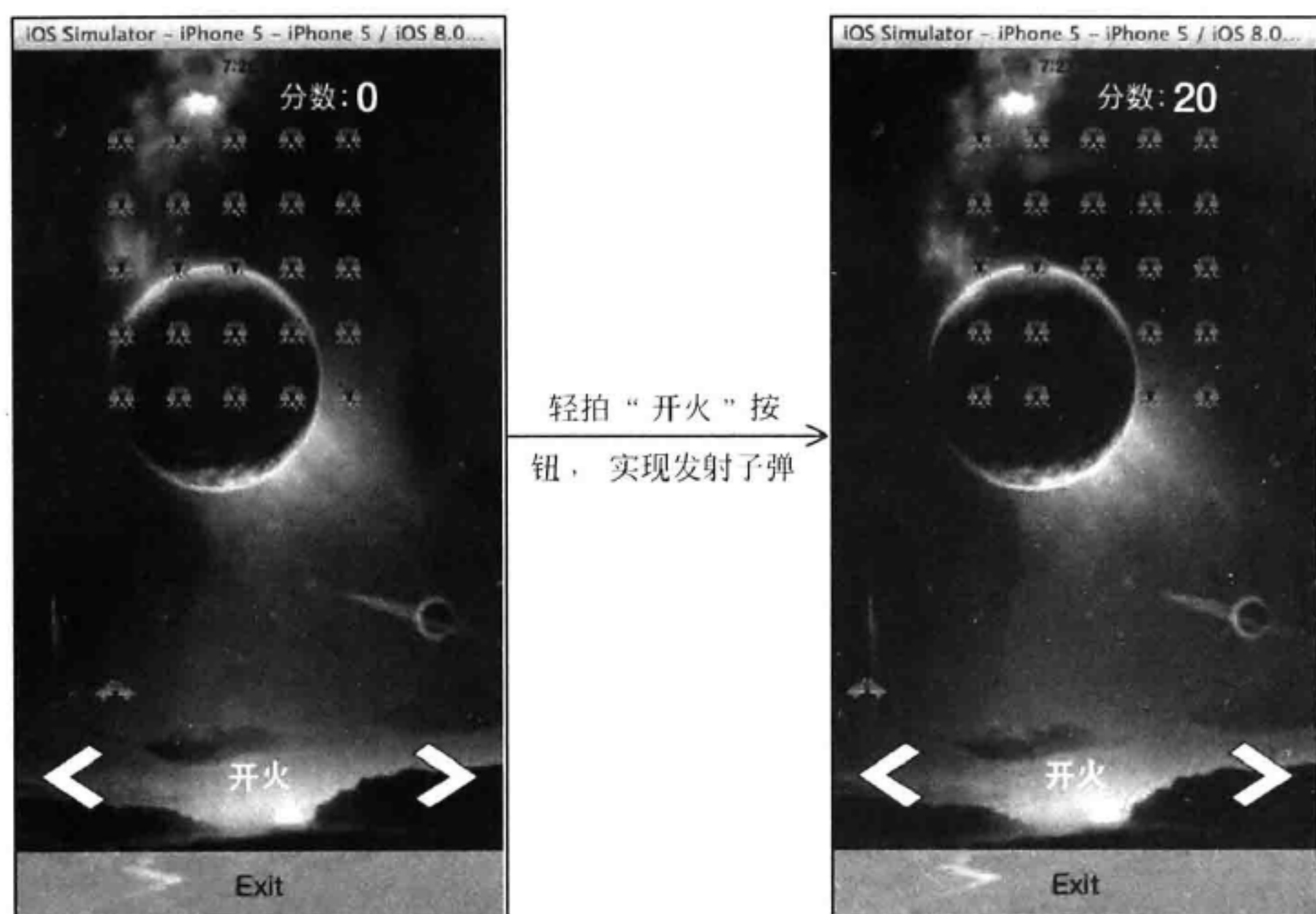


图 7.27 运行效果 1

当敌人的子弹击中飞船后，弹出警告视图，当玩家轻拍警告视图中的“是”按钮后实现分数的保存。打开分数榜后，会看到如图 7.28 所示的效果。



图 7.28 运行效果 2

当玩家轻拍警告视图中的“否”按钮后不会实现分数的保存。打开分数榜后，会看到如图 7.29 所示的效果。



图 7.29 运行效果 3

需要注意的是，在图 7.28 中可以看到，显示分数的表单元格既显示了分数也显示了时间，并且分数和时间在同一个表单元格中。其实，一个表单元格并不是只有一个标签，而是有两个标签，这两个 UILabel 分别为 textLable 和 detaiTextLable。其中，textLable 标签为主标签，detaiTextLable 为副标签。由于表单元格显示的格式的不同会导致这两个标签显示位置以及字体发生改变。其中表单元格的格式可以在创建时进行设置。其创建表单元格的语法形式如下：

```
UITableViewCell(style: UITableViewCellStyle, reuseIdentifier:String?)
```

其中，style 就是用来设置表格元格显示格式的。它的显示格式有 4 种，如下所述。

- 1. Default
该格式提供了一个简单的左对齐的文本标签 textLabel 和一个可选的图像 imageView。如果显示图像，那么图像将在最左边，如图 7.30 所示。
- 2. Subtitle
该格式与前一种相比，增加了对 detailTextLabel 的支持，该标签将会显示在 textLabel 标签的下面，字体相对较小，如图 7.31 所示。
- 3. Value1
该格式居左显示 textLabel，居右显示 detailTextLabel，且字体较小。该格式不支持图像，如图 7.32 所示。

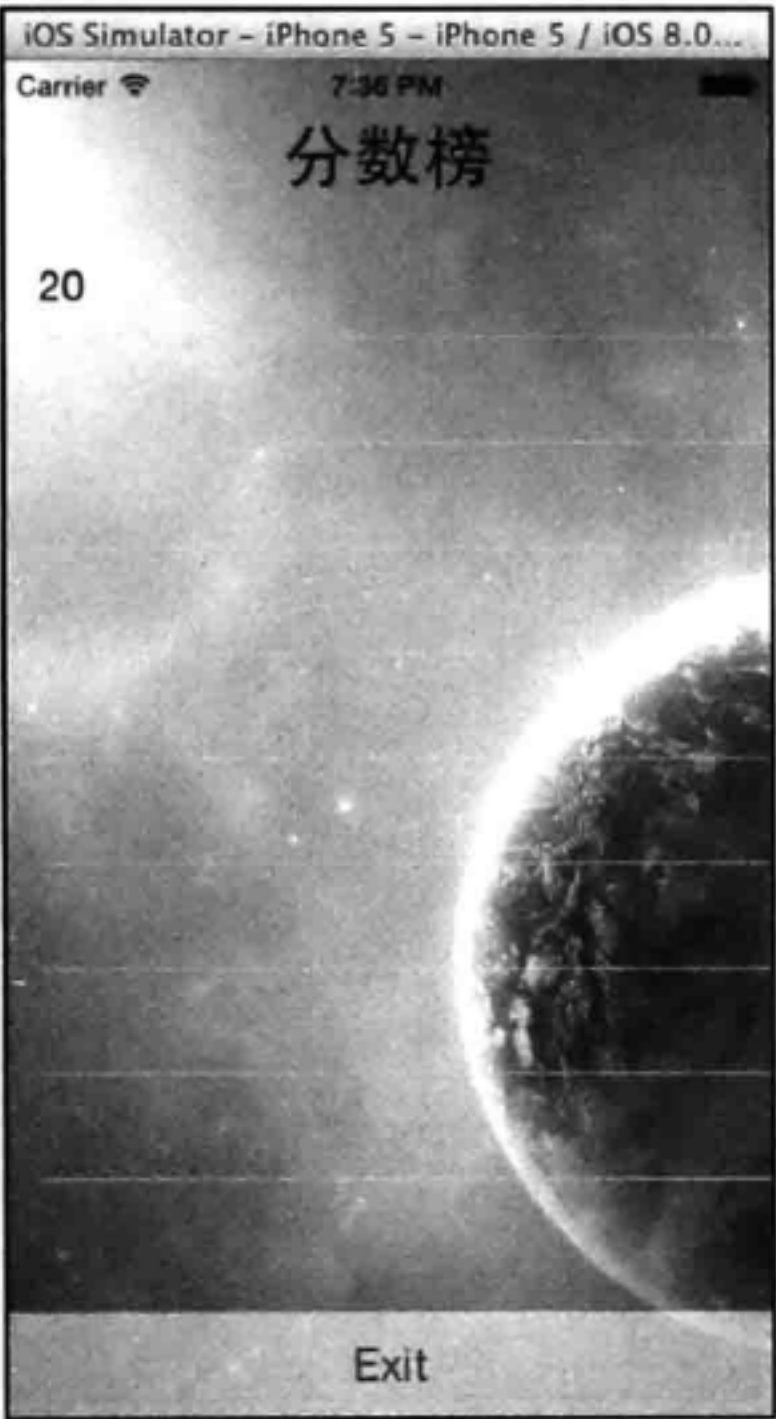


图 7.30 格式 1

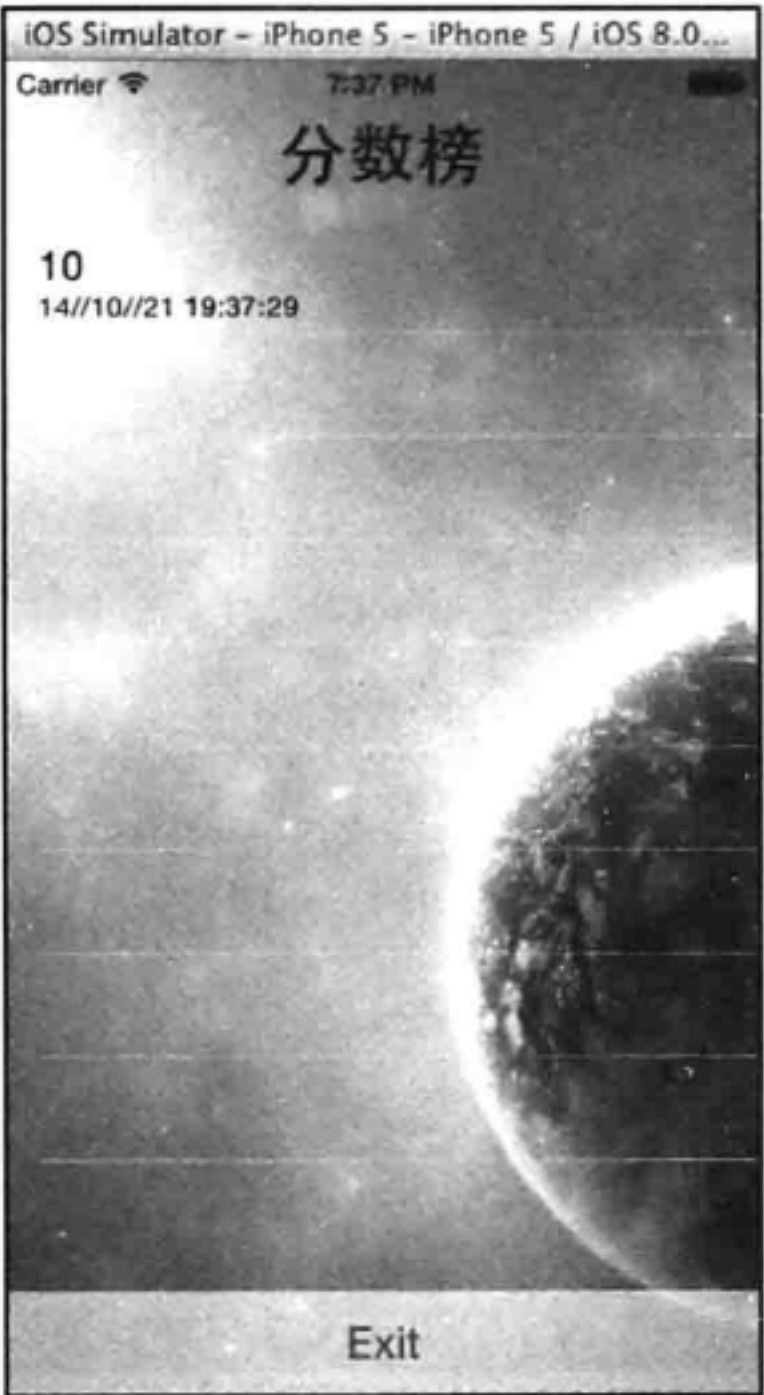


图 7.31 格式 2

4. Value2

该格式居左显示一个小型蓝色主标签 `textLabel`，在其右边显示一个小型黑色副标题详细标签 `detailTextLabel`。该格式不支持图像，如图 7.33 所示。

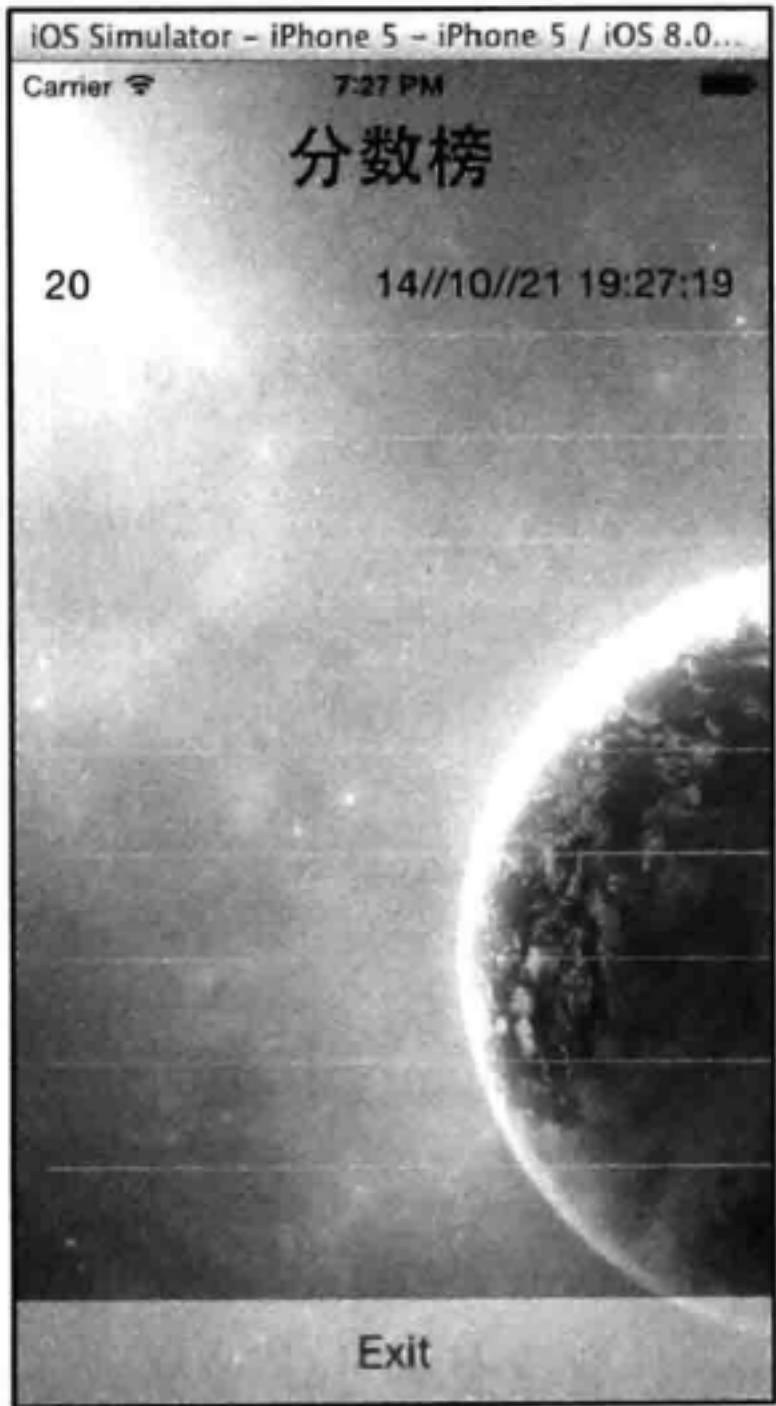


图 7.32 格式 3

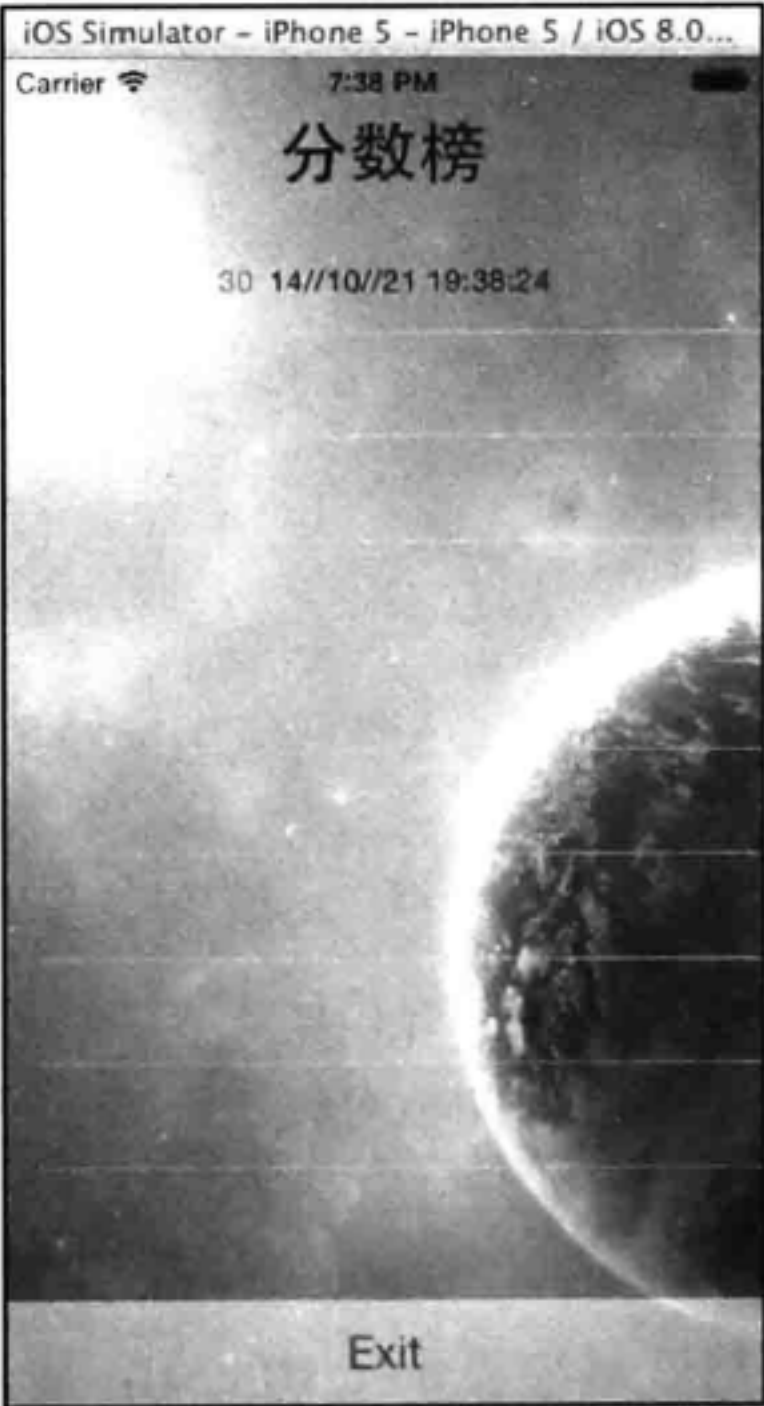


图 7.33 格式 4

7.11 场景切换

在此游戏中需要将所有的场景进行切换，具体的切换关系如表 7-5 所示。

表 7-5 场景切换

对 象	切 换 到
Play Game按钮	射击游戏的界面
Score List按钮	分数榜的界面
射击游戏界面中的Exit按钮	主菜单的界面
分数榜界面中的Exit按钮	
全部歼灭敌人界面中的“重新战斗”按钮	

最后画布中的效果如图 7.34 所示。

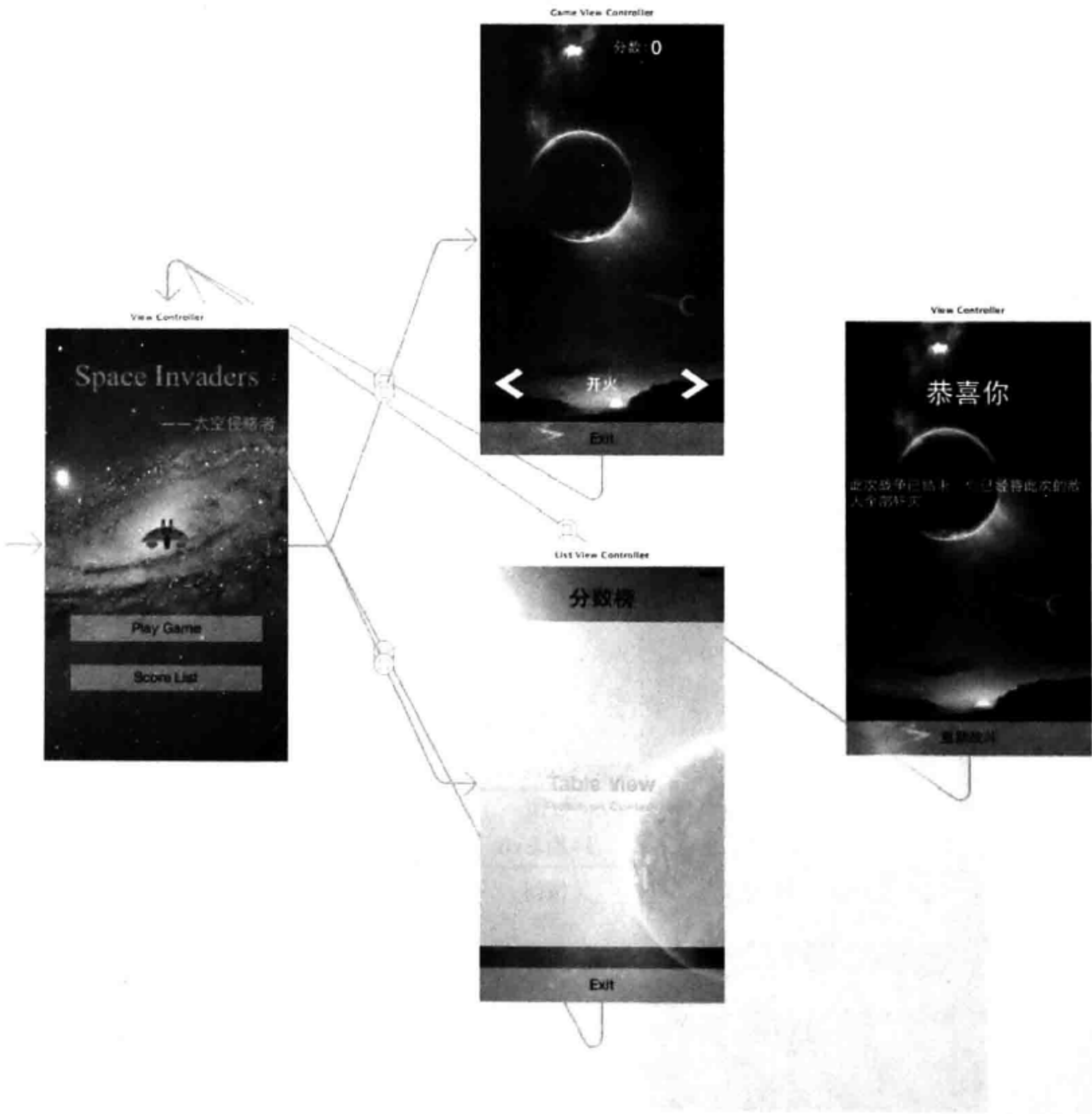


图 7.34 画布效果

注意：在实现场景切换前，首先需要单击实现主菜单的视图控制器，在此视图控制器中，选择界面上方的 Dock 中的 View Controller 图标，在工具窗口中的 Show the Attributes inspector 选项，即属性查看器选中，找到 Is Initial View Controller 复选框，将其选中，使此视图控制器成为初始视图控制器。

此时运行程序，可以看到以下的效果。当玩家在主菜单中轻拍 Play Game 按钮，可以进入射击游戏的界面。当玩家轻拍 Exit 按钮，可以返回到主菜单，效果如图 7.35 所示。

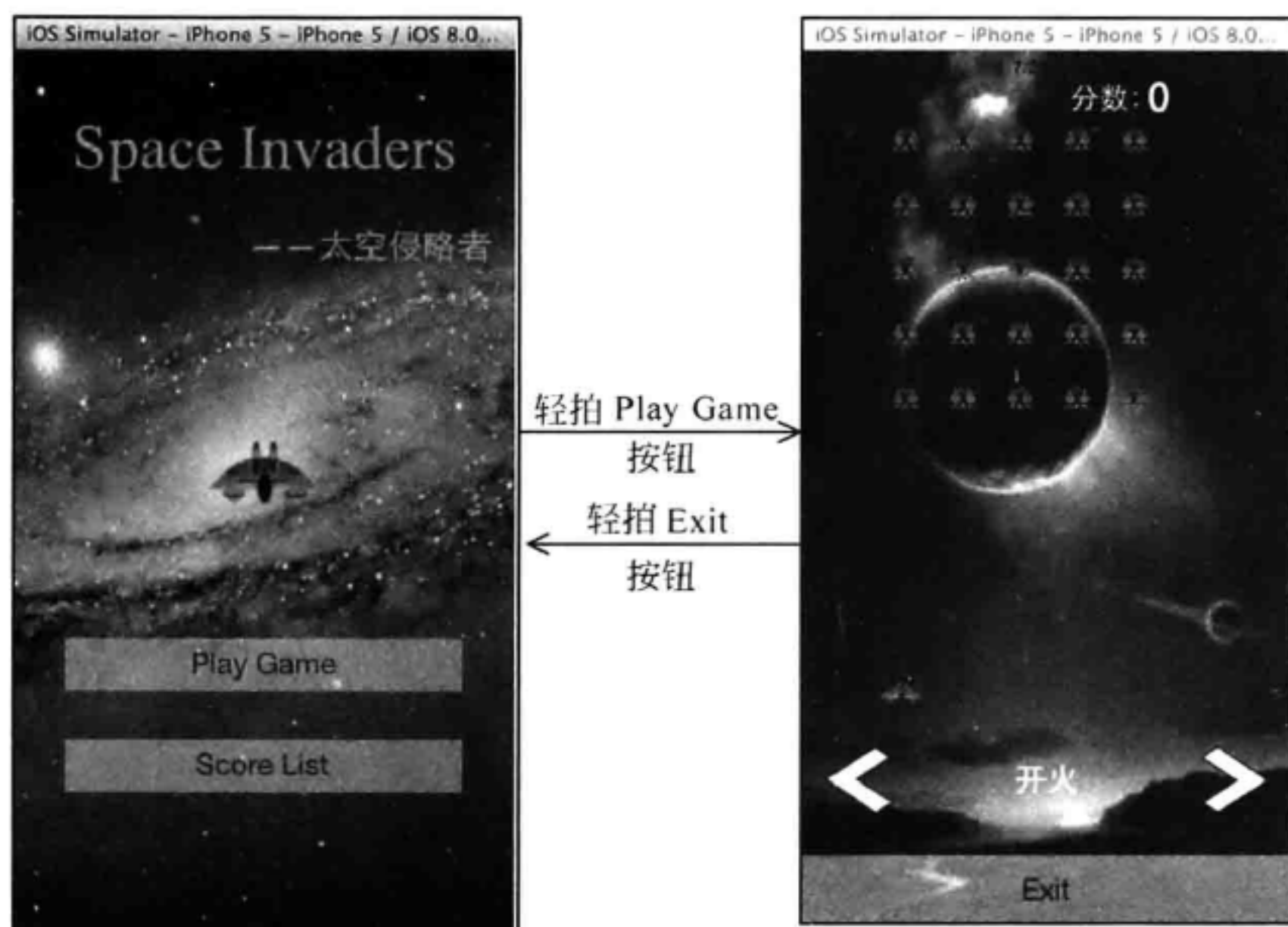


图 7.35 运行效果 1

当玩家在主菜单中轻拍 Score List 按钮，可以进入分数榜的界面。当玩家轻拍 Exit 按钮，可以返回到主菜单，效果如图 7.36 所示。

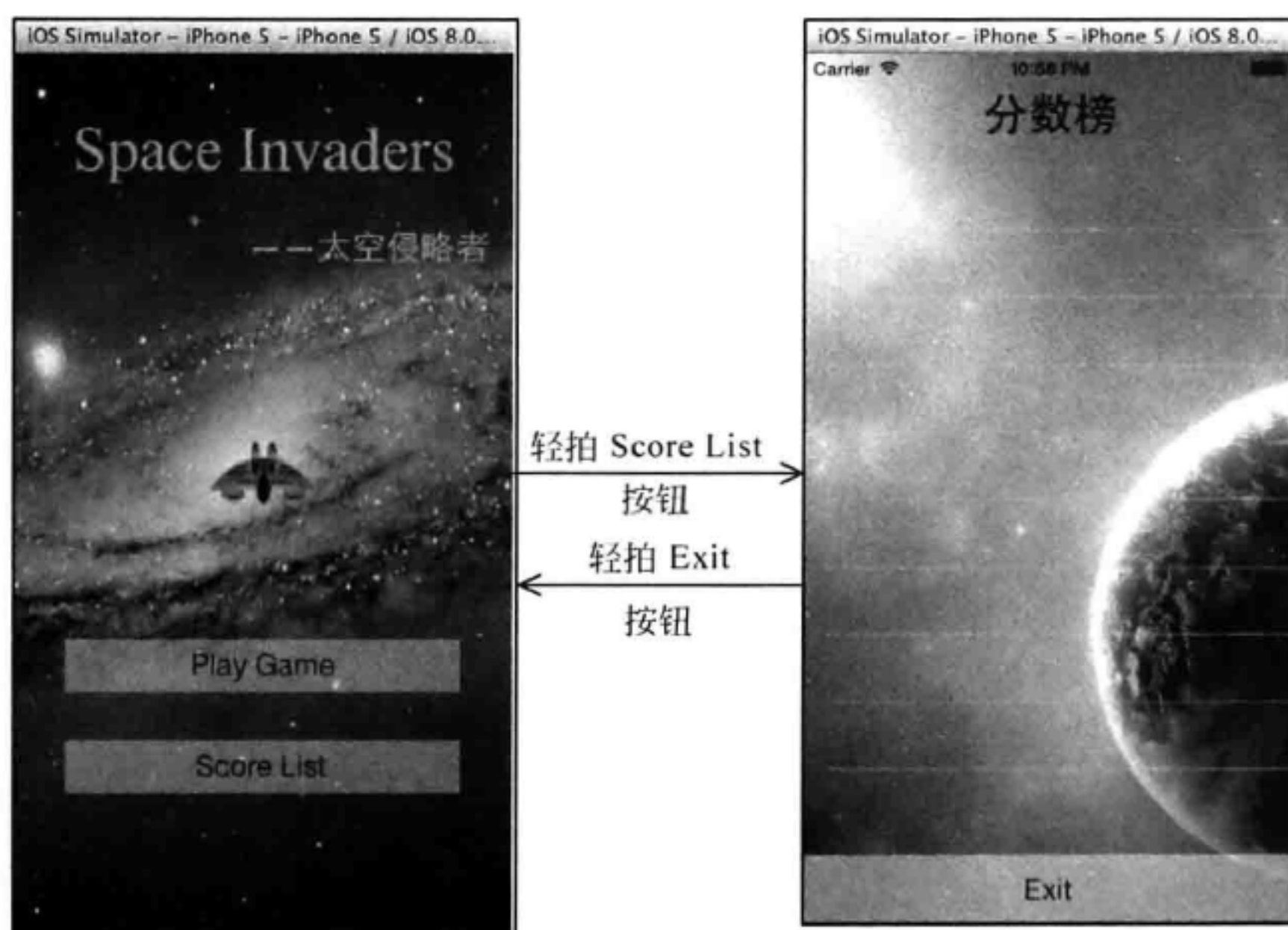


图 7.36 运行效果 2

当玩家歼灭全部敌人后，可以进入到歼灭全部敌人的界面。当玩家轻拍“重新战斗”按钮可以返回到主菜单，效果如图 7.37 所示。



图 7.37 运行效果 3

第 8 章 Simon 记忆游戏——音频引擎

Simon 记忆游戏是由 Simon says 演变而来的。Simon says 游戏是一个英国传统的儿童游戏。此游戏一般由 3 个或更多的人参加。其中，一个人充当 Simon，其他人必须根据情况对 Simon 宣布的命令做出不同反应。据心理学研究发现，这个游戏可以帮助儿童加强自我控制和对冲动行为的约束，所以深受大众的喜爱。很多的游戏开发商看到了此游戏的利益，就开发出了 Simon 记忆游戏。本章将讲解此游戏的实现。

8.1 游戏介绍

Simon 记忆游戏是一款简单但具有挑战性的游戏。它要求玩家需要记住所提示的颜色序列，然后按照之前的提示轻拍颜色。如果轻拍颜色的序列和提示的颜色序列相同，就会进入第二回合的记忆游戏中；否则，就重新开始游戏。这样一个游戏，包括以下几个模块。

1. 主菜单模块

在主菜单的界面上提供了两个菜单项，如图 8.1 所示。轻拍 Play Game 菜单项，进入游戏界面；轻拍 Game Description 菜单项，进入游戏介绍界面。

2. 游戏模块

游戏模块提供了游戏的界面，如图 8.2 所示。玩家可以在此界面中进行一些操作，并且做出相应的响应。



图 8.1 主菜单模板

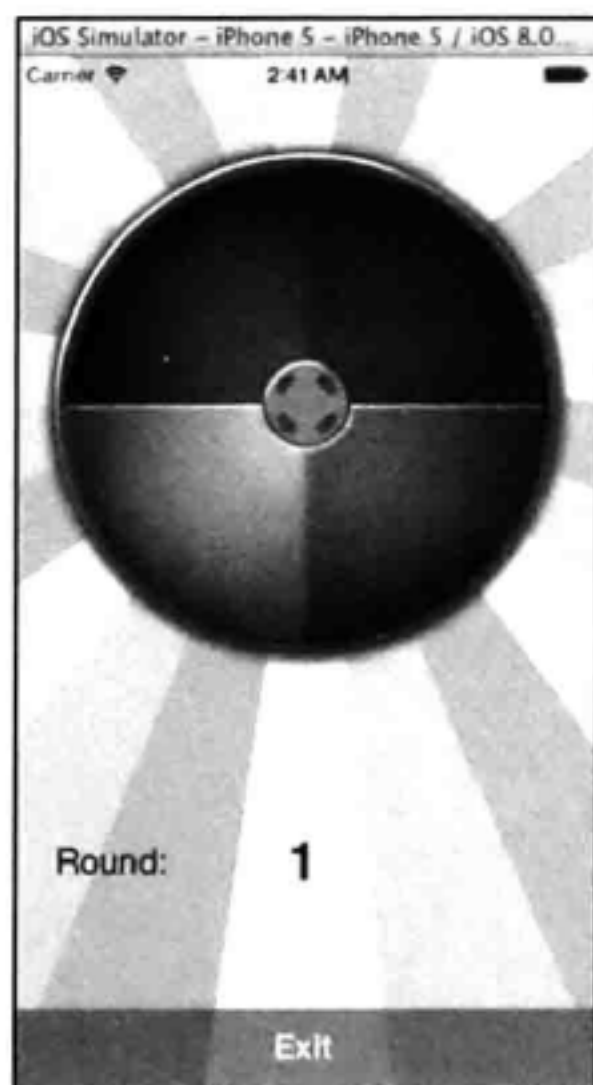


图 8.2 游戏模板

3. 游戏介绍模块

游戏介绍模块提供了游戏玩法的说明，如图 8.3 所示。



图 8.3 游戏介绍模块

8.2 开发游戏之前的准备工作

在开发 Simon 记忆游戏之前，需要做一些准备工作，这些准备工作介绍如下。

1. 创建项目

创建一个 Single View Application 模板类型的项目，命名为 Simon Memory Game。

2. 添加图像

添加图像 background.png、blue.png、empty.png、gameboard.png、green.png、logo.png、red.png 和 yellow.png 到创建项目的 Supporting Files 文件夹中，如图 8.4 所示。

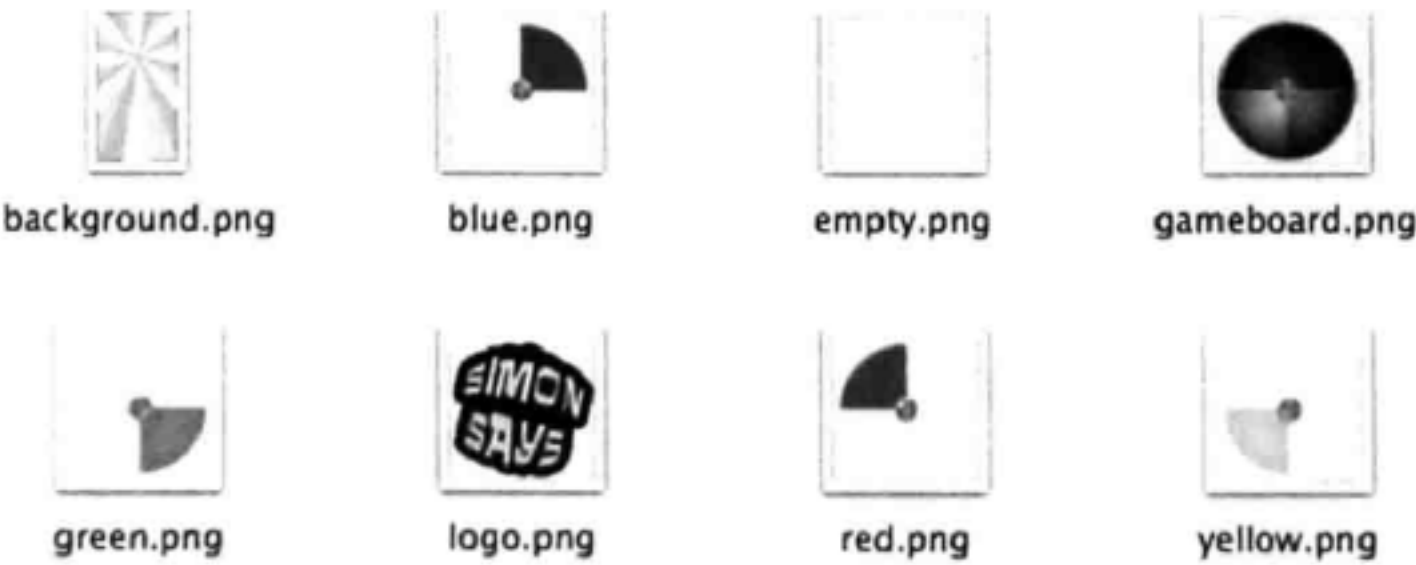


图 8.4 需要添加的图像

3. 添加音频文件

添加音频文件 blue.wav、Energy.wav、error.wav、green.wav、music.mp3 和 yellow.wav 到创建项目的 Supporting Files 文件夹中。

注意：音频文件的添加和图像的添加是一样的。首先，右击 Supporting Files 文件夹，在弹出的快捷菜单中，选择 Add Files to "Simon Memory Game"...命令，弹出选择文件对话框，如图 8.5 所示。选择需要添加的音频文件后，单击 Add 按钮，此时音频文件就被添加到了 Supporting Files 文件夹中。

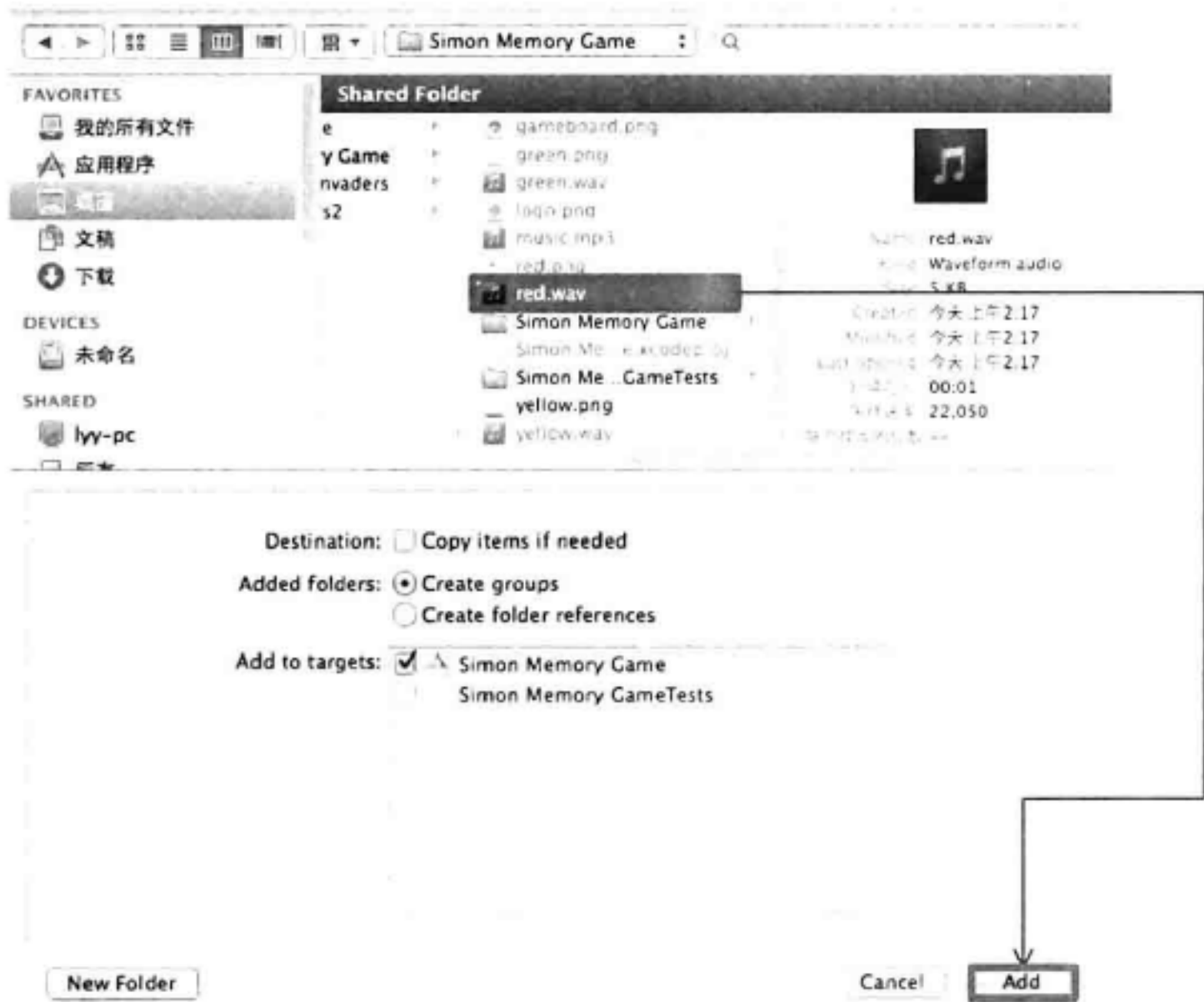


图 8.5 添加音频文件

4.添加框架

在本章的游戏中，我们需要使用到音频的播放。实现音频播放的类包含在 AVFoundation.framework 中，由于在创建的项目中没有此框架，所以需要将此框架添加到创建的项目中。具体的添加步骤介绍如下。

(1) 选择导航窗口中的项目名称，如图 8.6 所示。

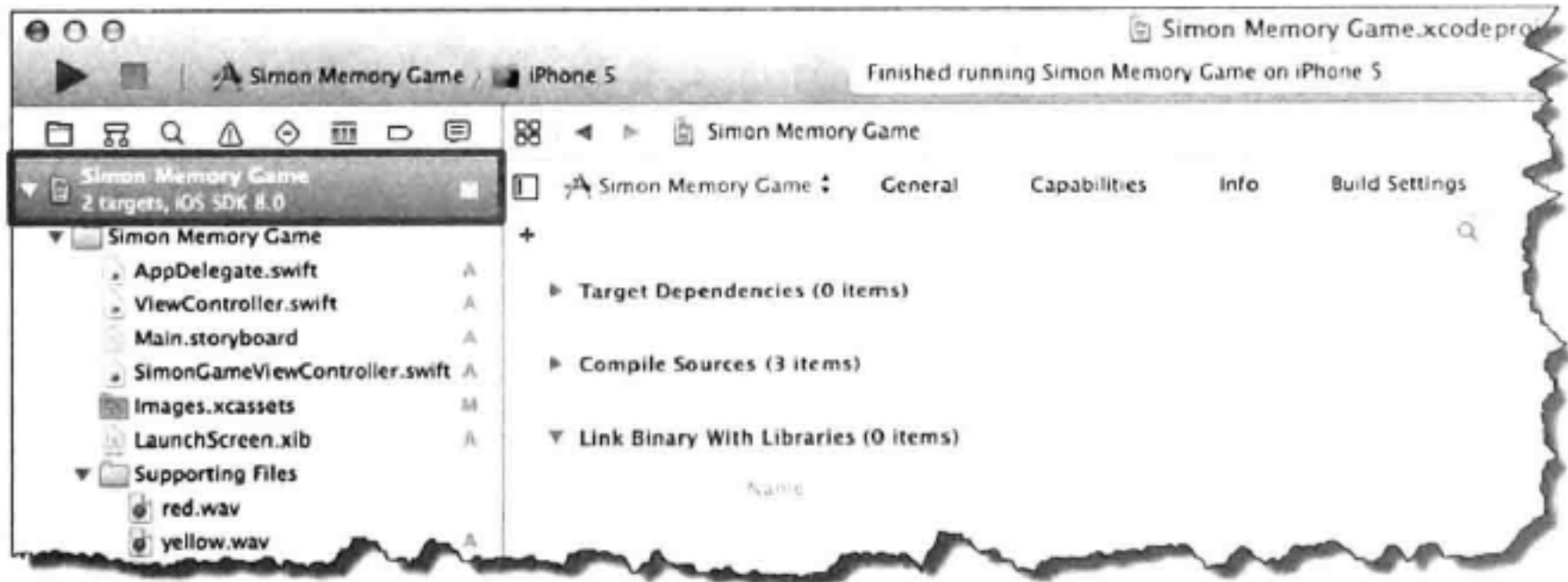


图 8.6 操作步骤 1

(2) 选择项目名称后，会打开目标窗口，如图 8.7 所示。

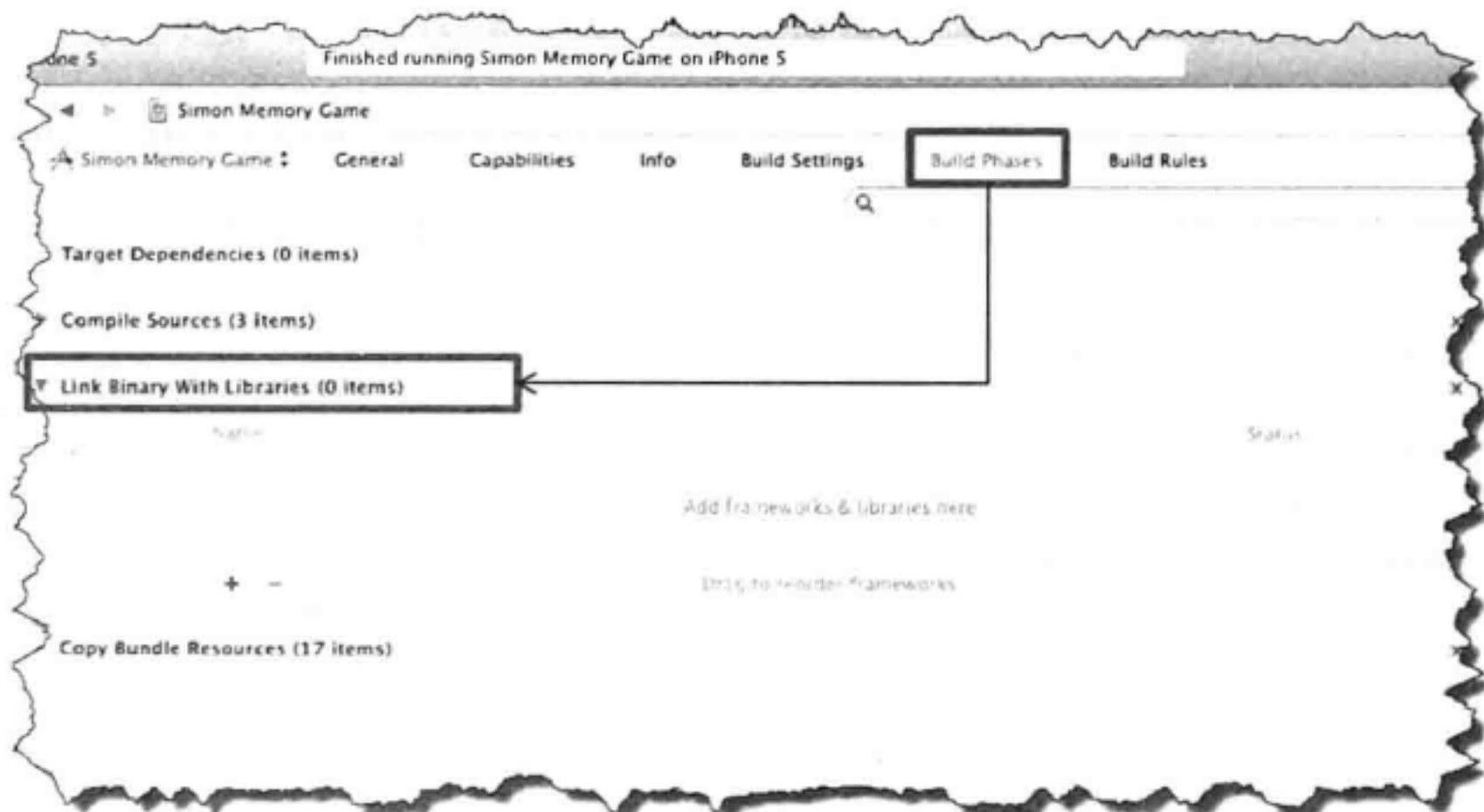


图 8.7 操作步骤 2

(3) 选择目标窗口中的 Build Phases 选项。打开此选项中的内容。在打开的内容中将 Link Binary With Libraries(0 items)打开，会看到一个加号按钮，如图 8.8 所示。



图 8.8 操作步骤 3

(4) 选择加号按钮后，会弹出一个 Choose frameworks and libraries to add 对话框，如图 8.9 所示。

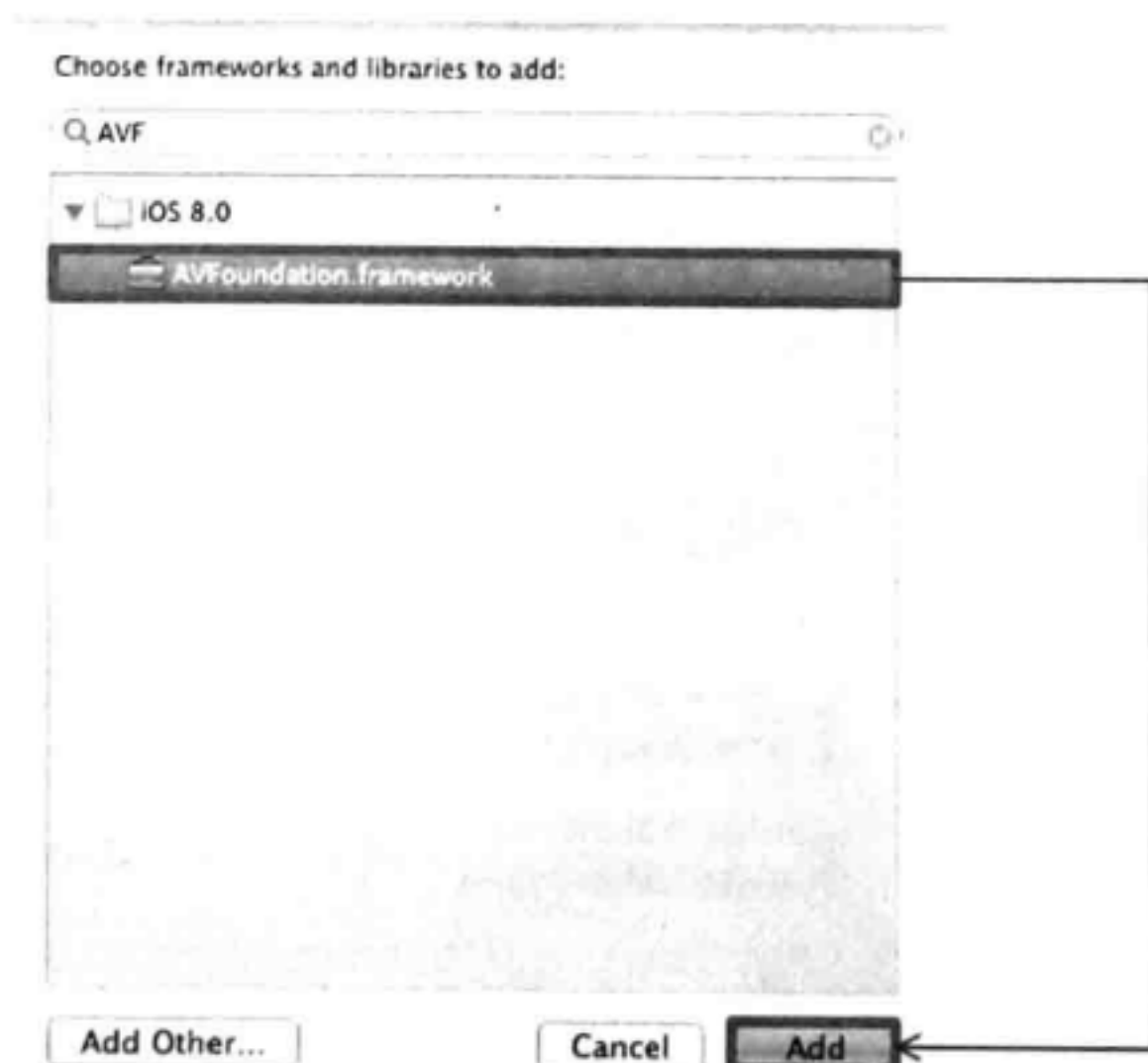


图 8.9 操作步骤 4

(5) 在对话框中找到 AVFoundation.framework 框架，然后单击 Add 按钮。此时，就会在 Link Binary With Libraries 中出现添加的框架，如图 8.10 所示。这表明此框架已经添加到了项目中。

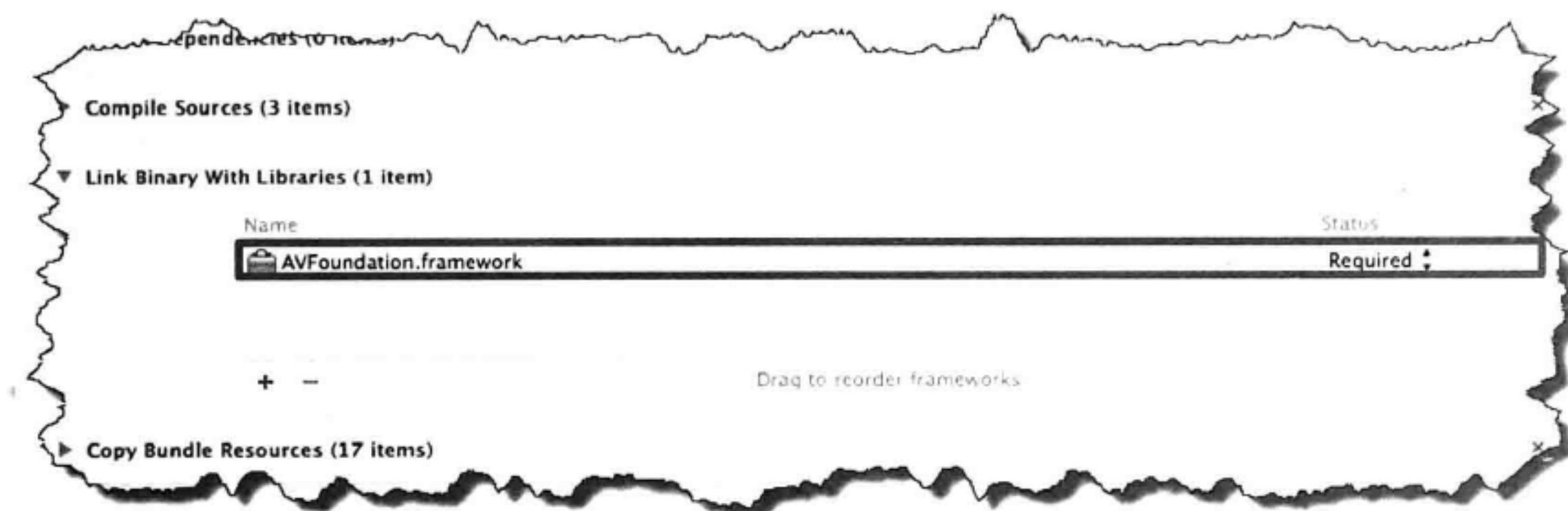


图 8.10 最后的效果

8.3 主菜单模块

每一个游戏都会有主菜单。Simon 记忆游戏也不例外。对于主菜单的菜单项，还是需要使用按钮来实现。本节将讲解主菜单的设计。

双击将 Main.storyboard 文件打开，对主菜单的界面进行设计，具体的操作步骤如下所述。

(1) 选择 Show the File inspector 选项，将 Interface Builder Document 中的 Opens in 改为 Xcode 5.1。

(2) 对在画板中原本存在的 View Controller 视图控制器的界面进行设计，效果如图 8.11 所示。

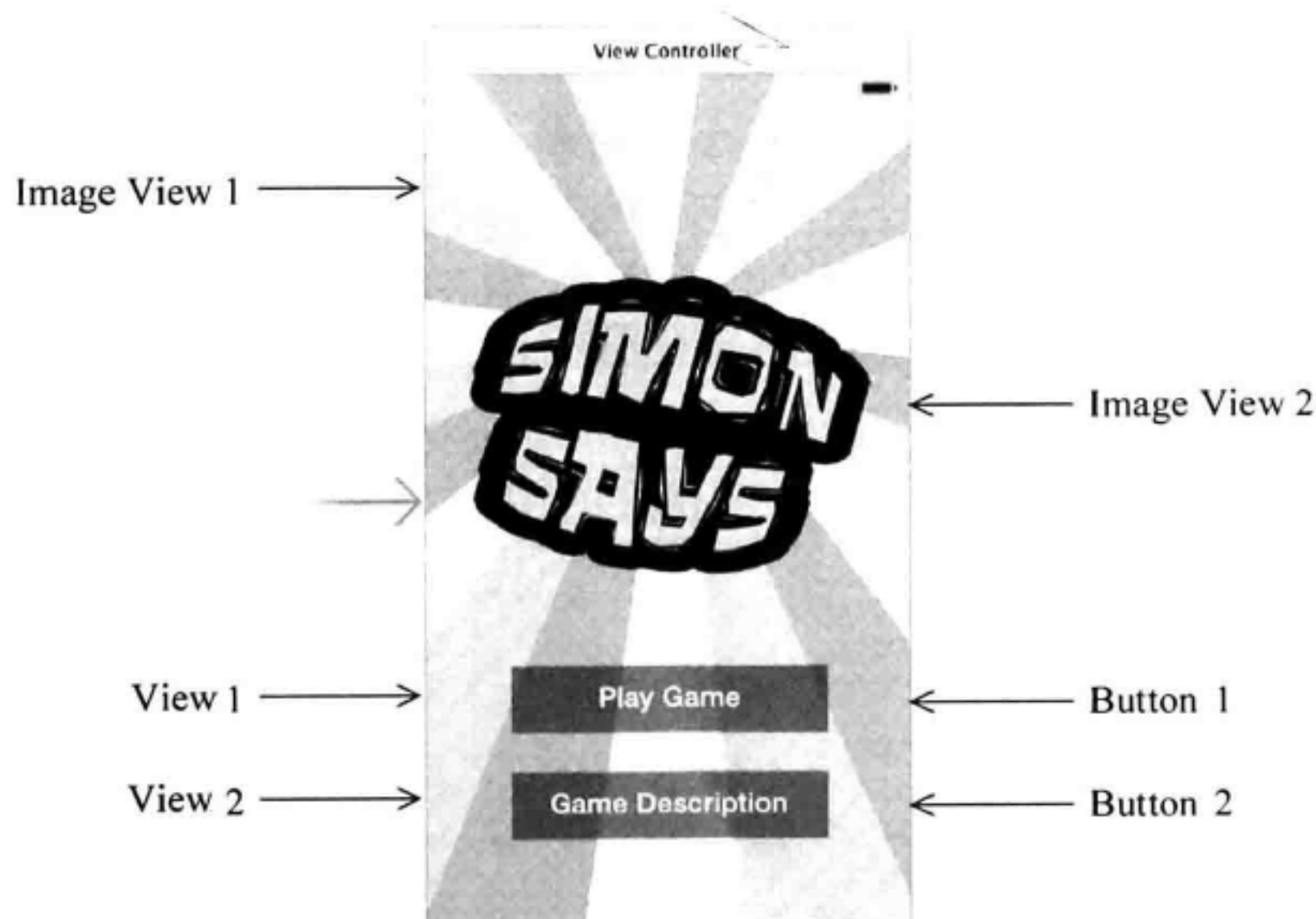


图 8.11 界面的效果

需要添加的视图对象，以及对它们的设置，如表 8-1 所示。

表 8-1 设置界面

视 图	设 置
Image View1	Image: background.png 位置和大小: (0, 0, 320, 568)
Image View2	Image: logo.png 位置和大小: (22, 122, 276, 218)
View1	Alpha: 0.6 Background: 深灰色 位置和大小: (56, 395, 209, 45)
Button1	Title: Play Game Font: System Bold 19.0 Text Color: 白色 位置和大小: (54, 7, 101, 30)
View2	Alpha: 0.6 Background: 深灰色 位置和大小: (56, 395, 209, 45)
Button2	Title: Game Description Font: System 19.0 Text Color: 黑色 位置和大小: (16, 7, 177, 30)

8.4 游 戏 模 块

本节将讲解有关游戏模块的一些操作，如准备工作和界面设计等。

8.4.1 准备工作

在设计游戏界面前，需要做一些准备工作，如为了便于代码的管理，将每一个界面实现的功能代码保存在一个文件中。

1. 创建文件

在创建的项目中需要创建一个 Swift File 模板类型的文件，命名为 SimonGameViewController。

2. 创建空类

打开创建的 SimonGameViewController.swift 文件，在此文件中创建一个基于 UIViewController 类的空类 SimonGameViewController，代码如下：

```
import UIKit
class SimonGameViewController: UIViewController {
}

```

该类用于编写实现 Simon 记忆游戏的代码。

8.4.2 界面设计

以下是对游戏界面进行设计的具体操作步骤。

(1) 在视图对象库中拖动 View Controller 视图控制器对象到画布中。

(2) 单击新添加的视图控制器，选择界面上方的 Dock 中的 View Controller 图标。在工具窗口中的 Show the Identity inspector 选项，即标示查看器中，将 Custom Class 下的 Class 设置为创建的 SimonGameViewController 类。这时，在画布中的这个视图控制器就变为了 Simon Game View Controller 视图控制器。

(3) 对 Simon Game View Controller 视图控制器的界面进行设计，效果如图 8.12 所示。

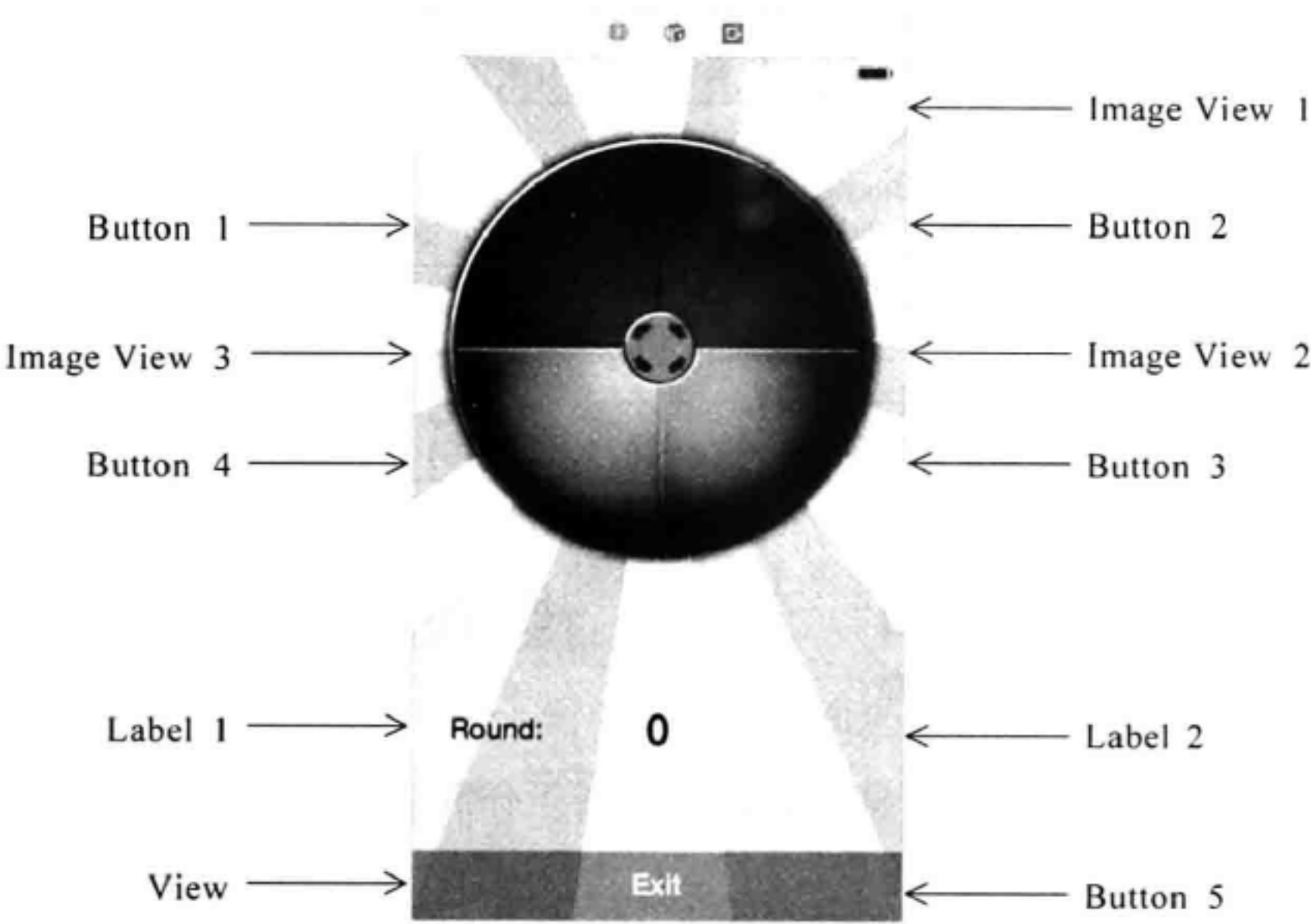


图 8.12 界面效果

需要添加的视图对象，以及对它们的设置，如表 8-2 所示。

表 8-2 设置界面

视 图	设 置
Image View1	Image: background.png 位置和大小: (0, 0, 320, 568)
Image View2	Image: gameboard.png 位置和大小: (10, 41, 300, 300)
Image View3	Image: empty.png 位置和大小: (10, 41, 300, 300) 与 SimonGameViewController.swift 文件声明并关联一个插座变量 highlightImageView
Button1	Title: (空) Tag: 1 位置和大小: (24, 54, 135, 136) 与 SimonGameViewController.swift 文件声明并关联一个动作 triggerButtonSound

续表

视 图	设 置
Button2	Title: (空) 位置和大小: (160, 54, 135, 136) Tag: 2 与 SimonGameViewController.swift 文件的动作 triggerButtonSound 进行关联
Button3	Title: (空) 位置和大小: (24, 191, 135, 136) Tag: 3 与 SimonGameViewController.swift 文件的动作 triggerButtonSound 进行关联
Button4	Title: (空) 位置和大小: (160, 191, 135, 136) Tag: 4 与 SimonGameViewController.swift 文件的动作 triggerButtonSound 进行关联
Label1	Text: Round: Font: System 19.0 位置和大小: (24, 432, 79, 21)
Label2	Text: 0 Font: System Bold 29.0 Alignment: 居中 位置和大小: (65, 404, 190, 77) 与 SimonGameViewController.swift 文件声明并关联一个插座变量 guessLabel
View	Alpha: 0.6 Background: 深灰色 位置和大小: (0, 523, 320, 45)
Button5	Title: Exit Font: System Bold 19.0 Text Color: 白色 位置和大小: (110, 7, 101, 30)

8.5 添加颜色提示序列

Simon 记忆游戏中也会有一个类似发出指令的人，此功能通过播放音频，以及调整图像视图的高亮来实现。本节将讲解添加音频播放以及颜色的提示。

8.5.1 添加提示声音

提示声音其实就是音频文件。音频文件的播放需要使用 AVFoundation.framework 框架中的 AVAudioPlayer 类来实现。添加提示声音的实现思路如下：当界面进入到游戏的界面后，定时器在 1 秒后进行声音的播放，播放的声音需要放在播放列表中。以下就是在本章游戏中添加提示声音的具体步骤。

(1) 导入 AVFoundation 类，代码如下：

```
import UIKit
import AVFoundation
```

(2) 声明并定义一些变量，代码如下：

```
var gameState:NSString=NSString()
var playListTimer:NSTimer?=nil
var guess:Int=1
var audioEffect:AVAudioPlayer=AVAudioPlayer()
```

 注意：在此代码中，gameState 用来存储游戏的状态，类似于一个状态机。

(3) 添加 viewDidLoad() 方法，在视图加载后调用，在此方法需要设置游戏的状态等。代码如下：

```
override func viewDidLoad() {
    self.gameState="Loading"
    self.newGame()
}
```

(4) 在 viewDidLoad() 方法中调用了 newGame() 方法。以下就是 newGame() 方法中需要编写的代码：

```
func newGame(){
    self.addAGuess()
}
```

(5) 在 newGame() 方法中调用了 addAGuess() 方法，在 addAGuess() 方法中需要实现随机数的发生，以及定时器的创建。代码如下：

```
func addAGuess(){
    self.gameState="GameTurn"
    //产生随机数
    var a=arc4random()
    var b=a%4
    var c=b+1
    var randEnemy:Int=Int(c)
    var soundNo:NSNumber=(NSNumber.numberWithInt(Int32(randEnemy)))
    guess=soundNo
    //创建定时器
    self.playListTimer=NSTimer.scheduledTimerWithTimeInterval(1, target:
self, selector: Selector("playSoundList"), userInfo: nil, repeats: false)
}
```

(6) playListTimer 定时器会在每隔 1 秒后调用一次 playSoundList() 方法。以下是此方法的实现代码：

```
func playSoundList(){
    var guessNo: NSNumber=guess
    self.playSound(guessNo.integerValue)
}
```

(7) 在 playSoundList() 方法中调用了 playSound() 方法，此方法实现的功能是使用 switch/case 语句判断音频文件名称以及 playSoundEffect(effectName:NSString) 方法的调用。代码如下：

```
func playSound(tag:Int){
```



```

var soundName:NSString=""
switch(tag){
    case 1:
        soundName="red"
    case 2:
        soundName="blue"
    case 3:
        soundName="yellow"
    case 4:
        soundName="green"
    default:
        soundName="error"
}
self.playSoundEffect(soundName)
}

```

(8) 添加 playSoundEffect(effectName:NSString)方法,在此方法中实现获取相应的音频文件后实现播放,从而实现提示功能。代码如下:

```

func playSoundEffect(effectName:NSString){
    //获取路径
    var path=NSBundle.mainBundle().pathForResource(effectName, ofType:
    "wav")
    var pathURL:NSURL=NSURL.fileURLWithPath(path!)
    audioEffect=AVAudioPlayer(contentsOfURL: pathURL,error: nil)
    //实例化对象 audioEffect
    audioEffect.numberOfLoops=0
    audioEffect.prepareToPlay()
    audioEffect.play() //播放声音
}

```

回到 Main.storyboard 文件中。单击 Simon Game View Controller 视图控制器。在此视图控制器中,选择界面上方的 Dock 中的 Simon Game View Controller 图标。在工具窗口中的 Show the Attributes inspector 选项,即属性查看器中,找到 Is Initial View Controller 复选框,将其选中。此时 Simon Game View Controller 视图控制器就成为了初始视图控制器。此时运行程序,会看到如图 8.13 所示的效果。

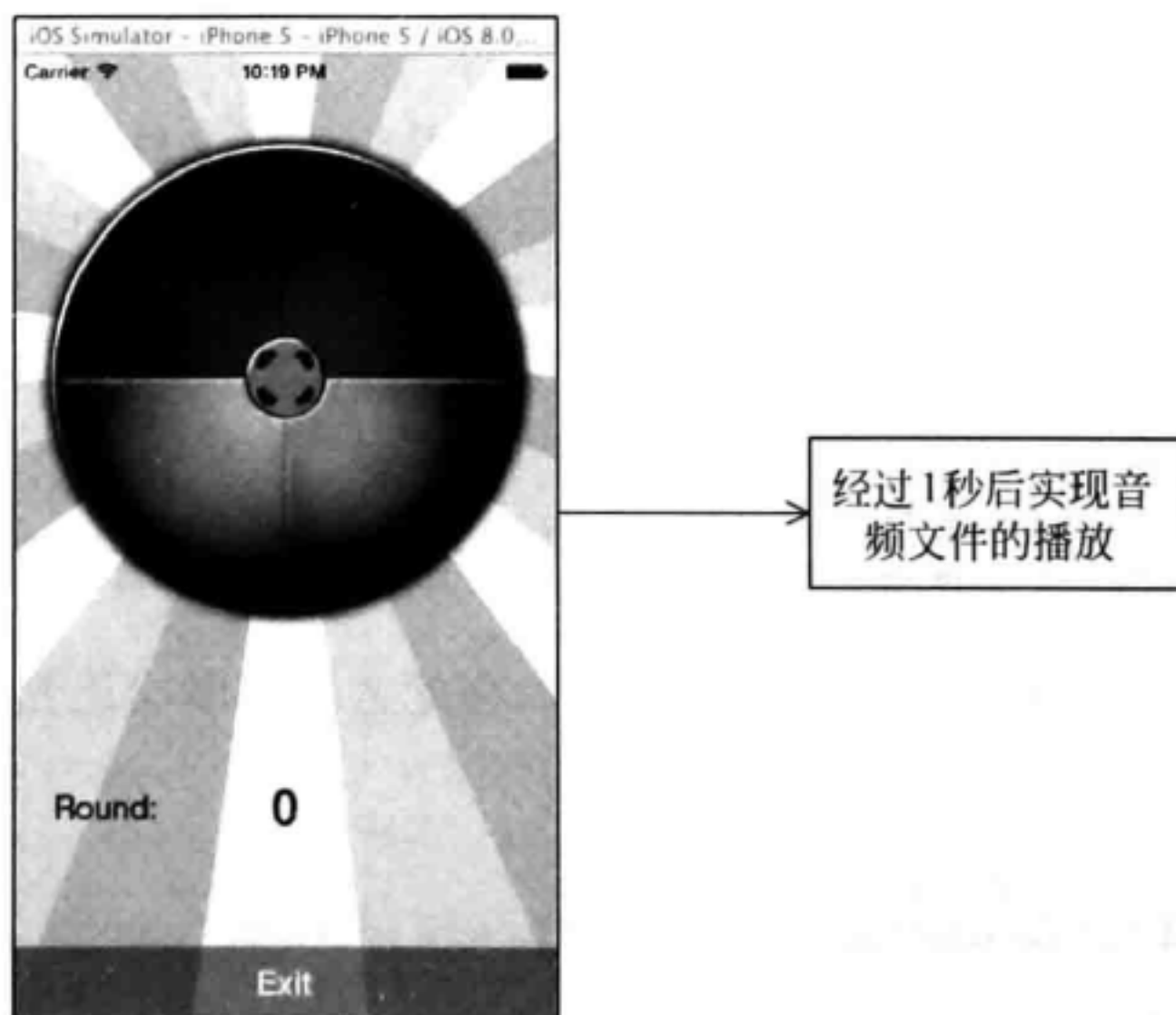


图 8.13 运行效果

8.5.2 添加颜色提示

如果声音的提示不是很明显，也可以为玩家添加颜色提示，即当音频文件播放至某一颜色时，在界面中对应的颜色就会变亮。此功能是通过改变图像视图的 `image` 属性来实现的。在 `playSound(tag:Int)` 方法中添加以下的代码：

```
self.highlightImageView.image=UIImage(named: "\(soundName).png")
self.playSoundEffect(soundName)
```

此时运行程序，会看到如图 8.14 所示的效果。

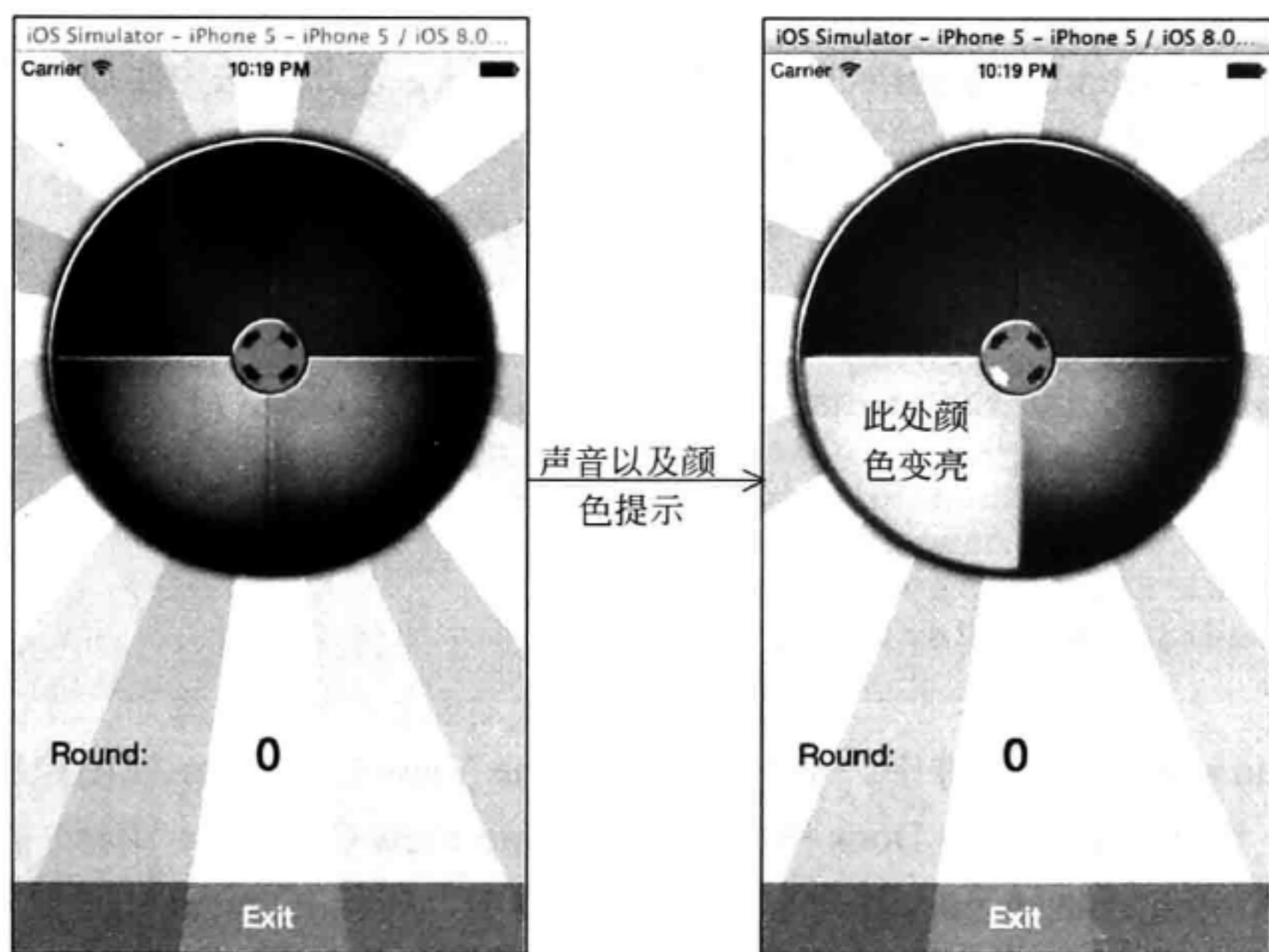


图 8.14 运行效果

在图 8.14 中可以看到，指定的颜色变亮了。但是当音频文件播放完毕后，此颜色还处于变亮的状态，如何将此颜色在音频文件播放完毕后，变为原先的颜色呢？以下就是此功能的具体实现。首先需要在 `playSoundList()` 方法中添加以下代码：

```
self.playSound(guessNo.integerValue)
self.gameState="Play"
NSTimer.scheduledTimerWithTimeInterval(0.5, target: self, selector:
Selector("hideButtonImage"), userInfo: nil, repeats: false)
```

然后需要添加一个 `hideButtonImage()` 方法，在此方法中添加让变亮的颜色变为原先的颜色。代码如下：

```
func hideButtonImage() {
    self.highlightImageView.image=UIImage(named: "empty.png")
}
```

此时运行程序，会看到如图 8.15 所示的效果。

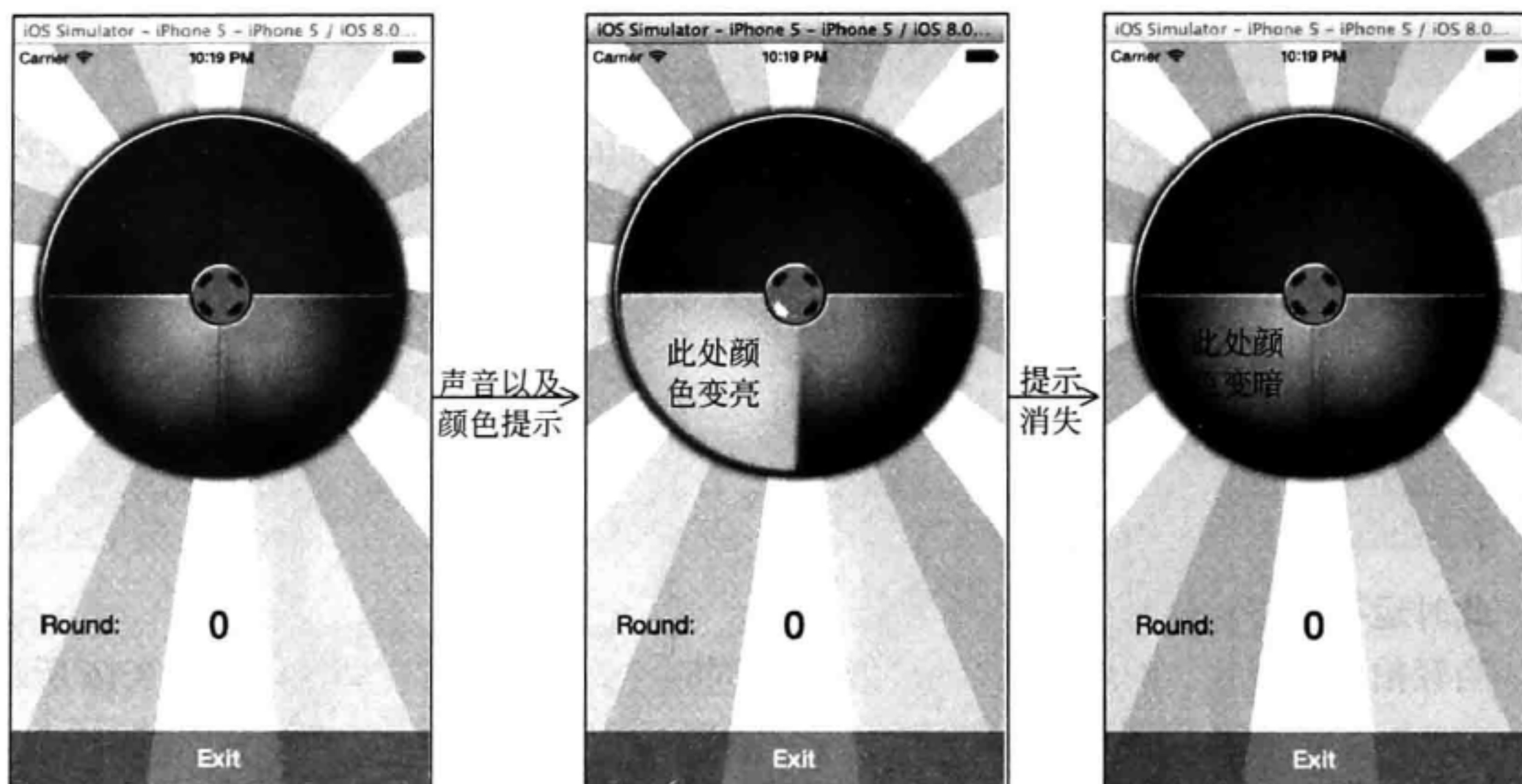


图 8.15 运行效果

8.6 玩家猜测

提示结束后，玩家就可以大显身手了，即实现猜测，按刚才提示颜色重新选择一次颜色。此功能需要借助在游戏界面中添加的 Button1~Button4 这 4 个按钮来实现（这 4 个按钮分别代表了 4 个颜色，即覆盖在图像视图上的颜色）。当按下这 4 个按钮中的其中一个后，将会触发判断颜色是否相同。简单一点说，就是判断提示颜色代表的数字是否和按钮的 tag 值相同。代码如下：

```
@IBAction func triggerButtonSound(sender: AnyObject) {
    //判断游戏的状态
    if(!(self.gameState.isEqualToString("Play"))){
        return
    }
    //判断提示颜色代表的数字是否和按钮的 tag 值相同
    if(guess==sender.tag){
        //相同的情况
        self.playSound(sender.tag) //播放指定的音频文件
        NSTimer.scheduledTimerWithTimeInterval(0.5, target: self, selector:
        Selector("hideButtonImage"), userInfo: nil, repeats: false)
        self.addAGuess() //继续猜测
    }else{
        //不同的情况
        var alert=UIAlertView()
        alert.title="对不起，你的颜色序列有误"
        alert.message="是否重新开始游戏"
        alert.addButtonWithTitle("是")
        alert.addButtonWithTitle("否")
        alert.show()
        alert.delegate=self
        self.playSound(-1)
    }
}
```



```
    }  
}
```

添加 `alertView(_alertView: UIAlertView,clickedButtonAtIndex buttonIndex: Int)`方法实现
对警告视图的响应，代码如下：

```
func alertView(_alertView: UIAlertView,clickedButtonAtIndex buttonIndex:  
Int){  
    var name:NSString=_alertView.buttonTitleAtIndex(buttonIndex)  
    if(name.isEqualToString("是")){  
        self.newGame()  
    }  
}
```

此时运行程序，会看到以下的效果。
当轻拍的按钮代表的颜色和提示的颜色相同时，就会继续进行猜测，如图 8.16 所示。

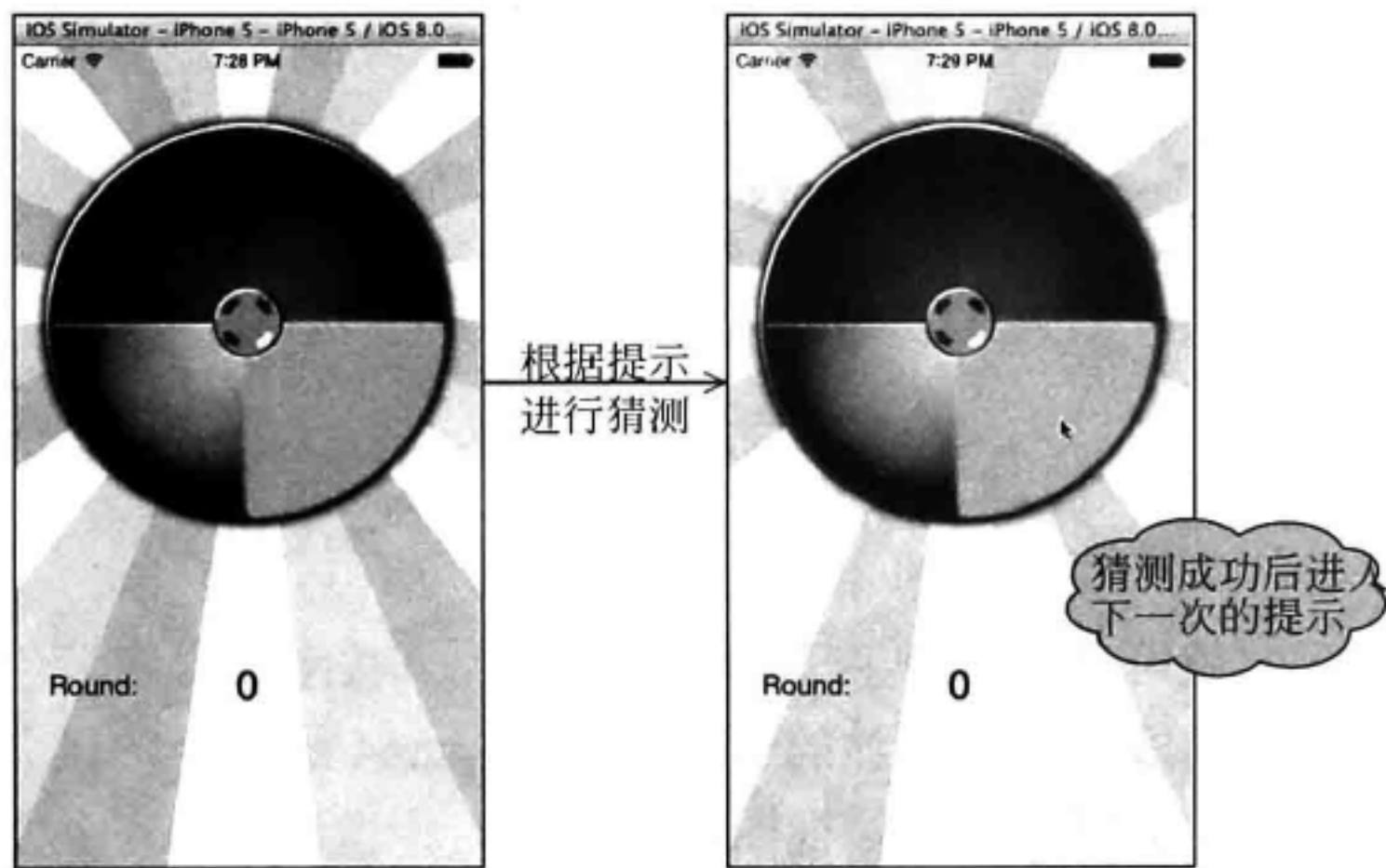


图 8.16 运行效果 1

当轻拍的按钮代表的颜色和提示的颜色不相同时，就会出现一个警告视图，如图 8.17 所示。

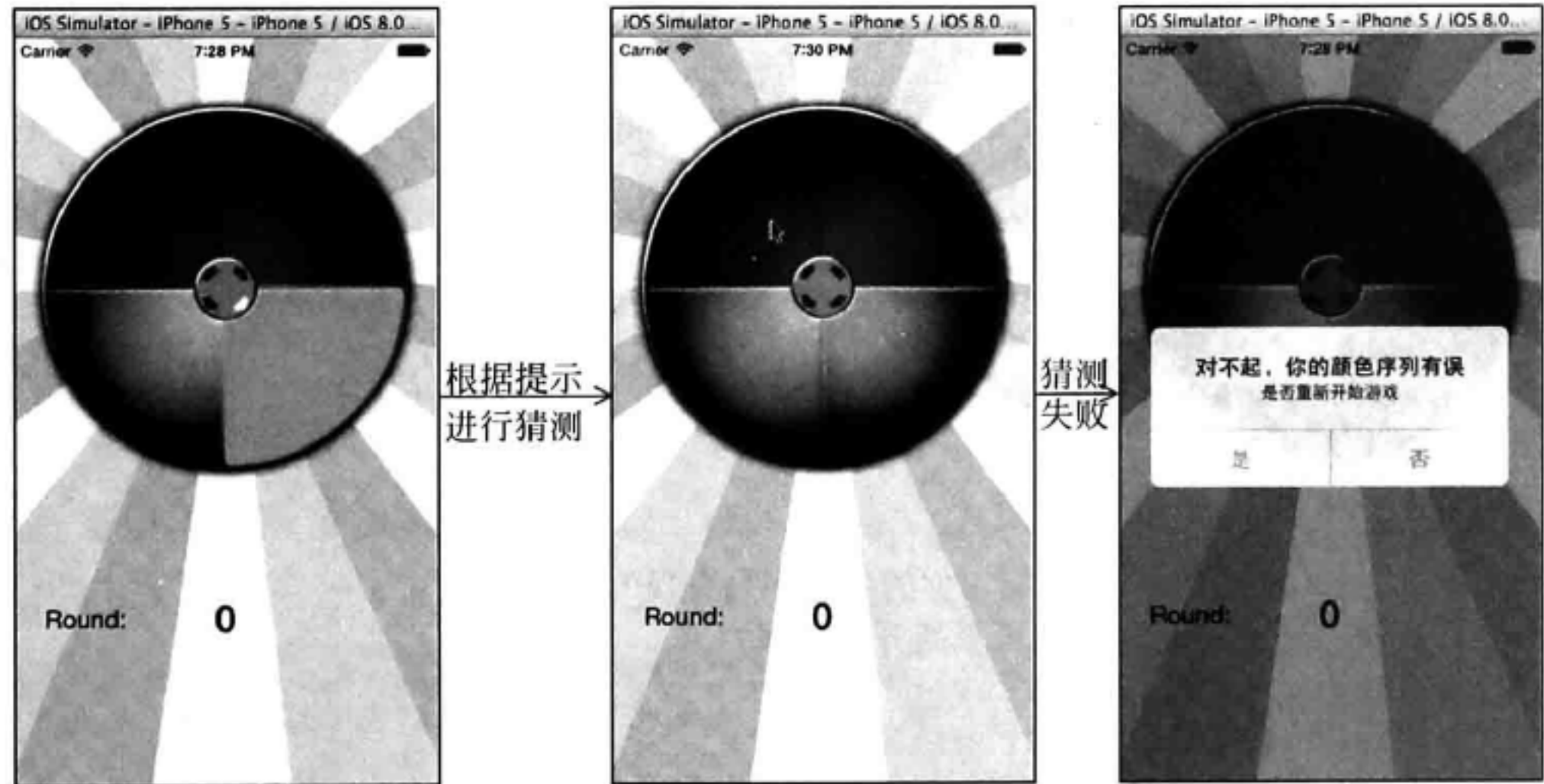


图 8.17 运行效果 2

当玩家轻拍警告视图中的“是”按钮后，就会重新开始游戏。当玩家轻拍“否”按钮后，就会退出游戏，如图 8.18 所示。



图 8.18 运行效果 3

8.7 添加背景音乐

为了让玩家可以放松心情，我们在游戏中添加了非常优美的背景音乐，背景音乐的添加以及播放和提示声音的添加和播放是一样的。以下就是添加背景音乐的具体实现。首先需要声明一个关于 AVAudioPlayer 的变量，代码如下：

```
var backgroundMusic:AVAudioPlayer?=nil
```

然后在 newGame()方法中添加一个可以调用 playMusic(musicName:NSString)方法的代码。代码如下：

```
self.playMusic("music")
```

最后在 playMusic(musicName:NSString)方法中实现背景音乐的播放。代码如下：

```
func playMusic(musicName:NSString){
    //判断 backgroundMusic 对象是否为空
    if(backgroundMusic==nil){
        varmusicFile:NSURL=NSBundle.mainBundle().URLForResource(musicName,
            withExtension: "mp3")! //获取路径
        backgroundMusic=AVAudioPlayer(contentsOfURL: musicFile,error: nil)
        //实例化对象
        //对对象 backgroundMusic 进行的设置
        backgroundMusic?.prepareToPlay()
        backgroundMusic?.numberOfLoops = -1
        backgroundMusic?.volume=0.25
    }
```

```

        backgroundMusic?.play() //背景音乐的播放
    }
}

```

此时运行程序，就可以听到优美的背景音乐了。

8.8 游戏模块的额外功能

本节将讲解关于游戏模块的一些额外功能，如显示游戏处于的关数和提高游戏的难度等内容。

8.8.1 显示游戏处于的关数

在以上运行的效果中，不管玩家挑战了多少次，都没有提示挑战的分数或者关数，使玩家缺少了趣味性，内心也得不到满足，也不可以截图向小伙伴们炫耀。这样一来，这款游戏就不是一款成功的游戏。为了弥补不足，我们在游戏中添加了关数。关数越多说明玩家的记忆能力越强。以下就是对游戏所处关数的实现。首先，声明一个变量，用来保存游戏的关数。代码如下：

```
var roundNumber:Int=0
```

然后在 `newGame()` 方法中添加以下代码，此代码实现的是刚进入游戏时，所处的是第一关。代码如下：

```

self.roundNumber=1
self.addAGuess()

```

在 `addAGuess()` 方法中添加以下代码，此代码实现的功能是在标签中显示关数。代码如下：

```

self.guessLabel.text="\ (self.roundNumber) "
self.playListTimer=NSTimer.scheduledTimerWithTimeInterval(1, target: self, selector: Selector("playSoundList"), userInfo: nil, repeats: false)

```

最后在 `triggerButtonSound` 动作中添加代码，即在成功实现一次记忆游戏的挑战后，为 `roundNumber` 变量加 1，代码如下：

```

if(guess==sender.tag){
    self.playSound(sender.tag)
    NSTimer.scheduledTimerWithTimeInterval(0.5, target: self, selector: Selector("hideButtonImage"), userInfo: nil, repeats: false)
    self.roundNumber++
    self.addAGuess()
}else{
    .....
}

```


此时运行程序，就可以看到如图 8.19 所示的效果了。

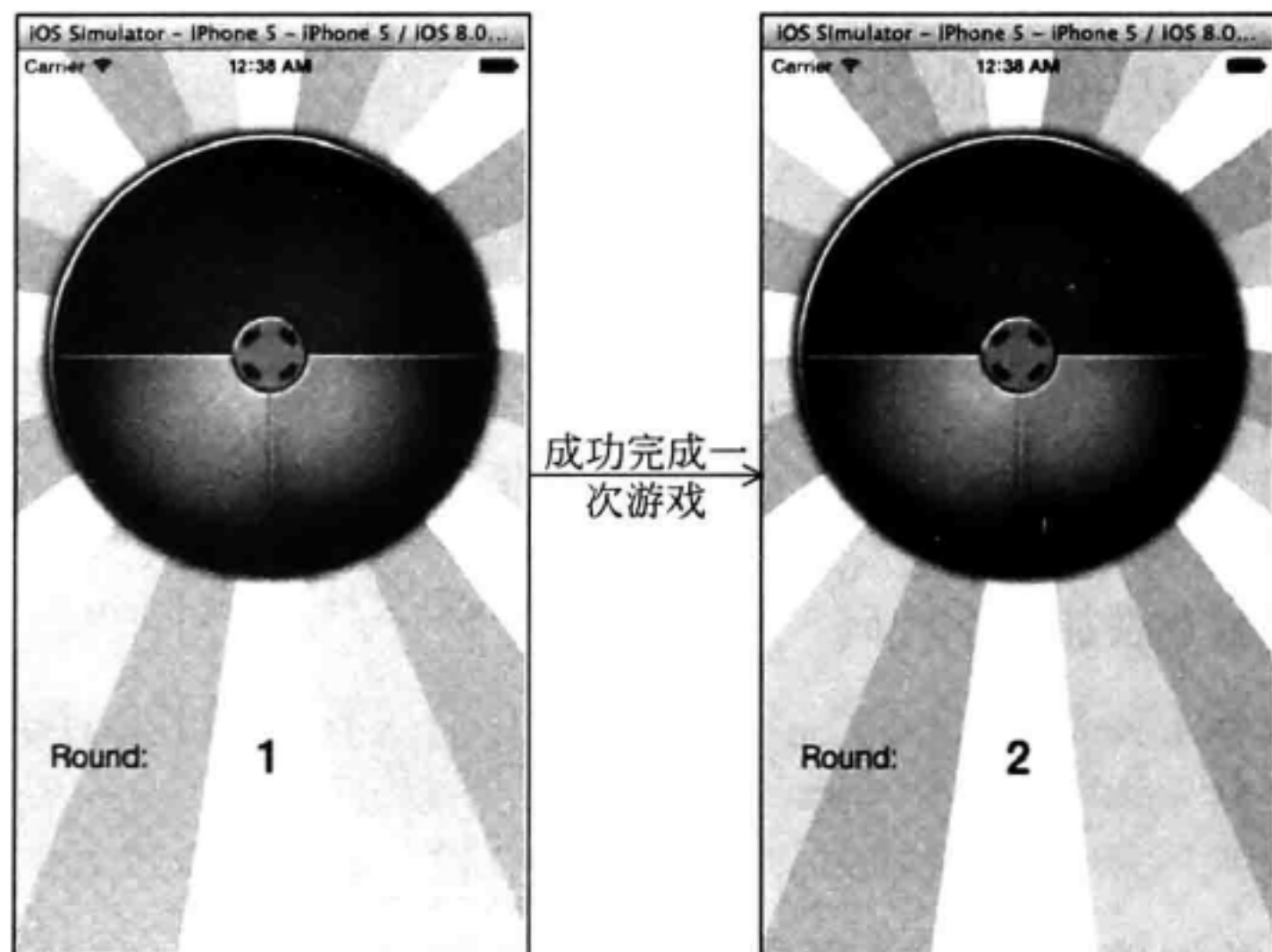


图 8.19 运行效果

📌注意：在标签中显示的数值表示的就是游戏处于的关数。只有当此关的游戏挑战成功后才可以进入到下一关。

8.8.2 提高游戏的难度

运行 8.8.1 节中的程序，我们不难看出，虽然关数有所增加，但是难度系数是一样的。在一款具有挑战性的游戏中，随着所处关数的增加，游戏的难度系数也会增加，让玩家有一种挑战性。在本小节中，我们将提升玩家的挑战性。我们在第一关会提示一个随机颜色，在第二关会连续提示两个随机颜色，在第 3 关会连续提示 3 个随机颜色，依此类推。随着游戏难度的不断增加，游戏中使用的提示颜色也会不断增加，这样一来使用 `guess` 变量来保存颜色代表的数字就不实用了，需要使用数组来实现。以下就是提高游戏难度的具体实现过程。

首先，需要删除以下的代码：

```
var guess:Int=1
```

其次添加以下的代码：

```
var guessNumber:Int?=nil
var currentSoundNo:Int?=nil
var guessList:NSMutableArray=NSMutableArray() //用来保存颜色序列
```

然后在 `newGame()` 方法中添加以下的代码：

```
self.roundNumber=1
guessNumber=0
guessNumber=0
```



```
self.guessList.removeAllObjects()
self.roundNumber=1
```

在 addAGuess()方法中添加以下的代码:

```
self.gameState="GameTurn"
currentSoundNo=0
guessNumber=0
```

在 addAGuess()方法中删除以下的代码:

```
guess=soundNo
```

在删除代码的地方添加以下代码:

```
var soundNo:NSNumber=(NSNumber.numberWithInt(Int32(randEnemy)))
self.guessList.addObject(soundNo)
self.guessLabel.text="\ (self.roundNumber) "
```

修改 playSoundList()中的方法, 代码如下:

```
func playSoundList(){
    var guessNo: NSNumber=self.guessList[currentSoundNo!] as NSNumber
    self.playSound(guessNo.integerValue)
    currentSoundNo!++
    //判断当前播放的个数是否小于数组 guessList 中的个数
    if(currentSoundNo<self.guessList.count){
        //创建定时器
        self.playListTimer=NSTimer.scheduledTimerWithTimeInterval(1,target:
        self, selector: Selector("playSoundList"), userInfo: nil, repeats:
        false)
    }else{
        self.gameState="Play"
        //创建定时器
        NSTimer.scheduledTimerWithTimeInterval(0.5, target: self, selector:
        Selector("hideButtonImage"), userInfo: nil, repeats: false)
    }
}
```

修改 triggerButtonSound 动作中的代码, 代码如下:

```
//判断 guessList 数组中指定的值是否和按钮的 tag 值相等
if(self.guessList[guessNumber!].integerValue==sender.tag){
    self.playSound(sender.tag)
    NSTimer.scheduledTimerWithTimeInterval(0.5, target: self, selector:
    Selector("hideButtonImage"), userInfo: nil, repeats: false)
    guessNumber!++
    //判断 guessNumber 是否和 roundNumber 相等
    if(guessNumber==self.roundNumber){
        self.roundNumber++
        self.addAGuess() //进入下一关
    }
}else{
    .....
}
```

此时运行程序, 就会看到随着关数的增加需要记忆的颜色也会相应的增加。

8.9 游戏介绍模块

本节将讲解关于游戏介绍界面的设计。在视图对象库中拖动 View Controller 视图控制器对象到画布中，然后对此视图控制器的界面进行设计，效果如图 8.20 所示。

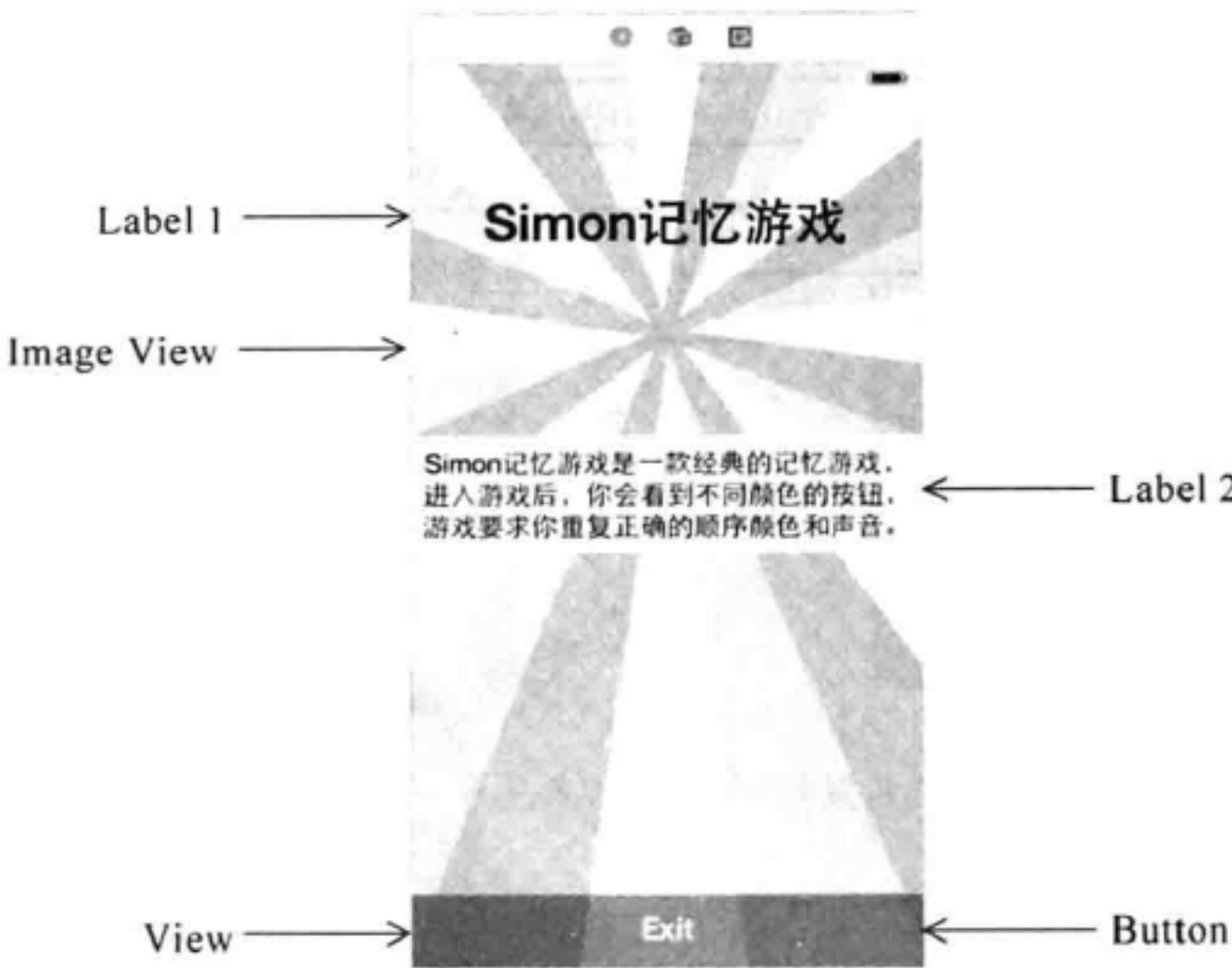


图 8.20 界面效果

需要添加的视图对象，以及对它们的设置，如表 8-3 所示。

表 8-3 设置界面

视 图	设 置
Image View	Image: background.png 位置和大小: (0, 0, 320, 568)
Label1	Text: Simon 记忆游戏 Font: System Bold 33.0 Alignment: 居中 位置和大小: (33, 80, 254, 42)
Label2	Text: Simon 记忆游戏是一款经典的记忆游戏，进入游戏后，你会看到不同颜色的按钮，游戏要求你重复正确的顺序颜色和声音。 Alignment: 居中 Lines: 3 位置和大小: (0, 239, 320, 79)
View	Alpha: 0.6 Background: 深灰色 位置和大小: (0, 523, 320, 45)
Button	Title: Exit Font: System Bold 19.0 Text Color: 白色 位置和大小: (110, 7, 101, 30)

8.10 场景切换

在此游戏中需要将所有的场景进行切换，具体的切换关系如表 8-4 所示。

表 8-4 场景切换

对 象	切 换 到
Play Game 按钮	游戏界面
Game Description 按钮	游戏介绍界面
游戏界面中的 Exit 按钮	主菜单界面
游戏介绍界面中的 Exit 按钮	

最后画布中的效果如图 8.21 所示。

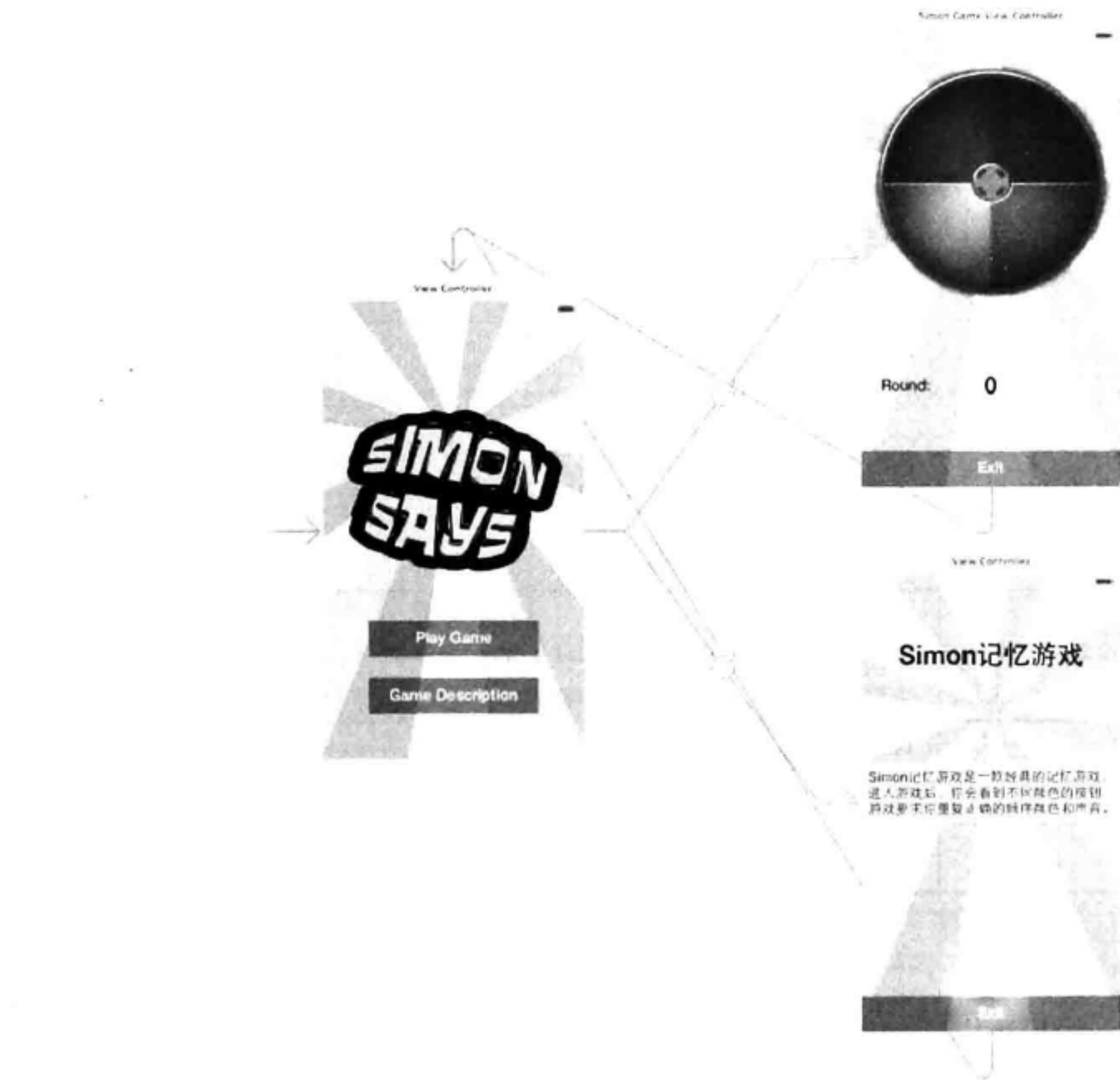


图 8.21 画布的效果

注意：在实现场景切换前，首先需要单击实现主菜单的视图控制器，在此视图控制器中，选择界面上方的 Dock 中的 View Controller 图标，在工具窗口中的 Show the Attributes inspector 选项，即属性查看器选中，找到 Is Initial View Controller 复选框，将其选中，使此视图控制器成为初始视图控制器。

此时运行程序，可以看到以下的效果。当玩家在主菜单中轻拍 Play Game 按钮，可以进入游戏界面。当玩家轻拍 Exit 按钮，可以返回到主菜单，效果如图 8.22 所示。



图 8.22 运行效果 1

当玩家在主菜单中轻拍 Game Description 按钮，可以进入游戏介绍界面。当玩家轻拍 Exit 按钮，可以返回到主菜单，效果如图 8.23 所示。

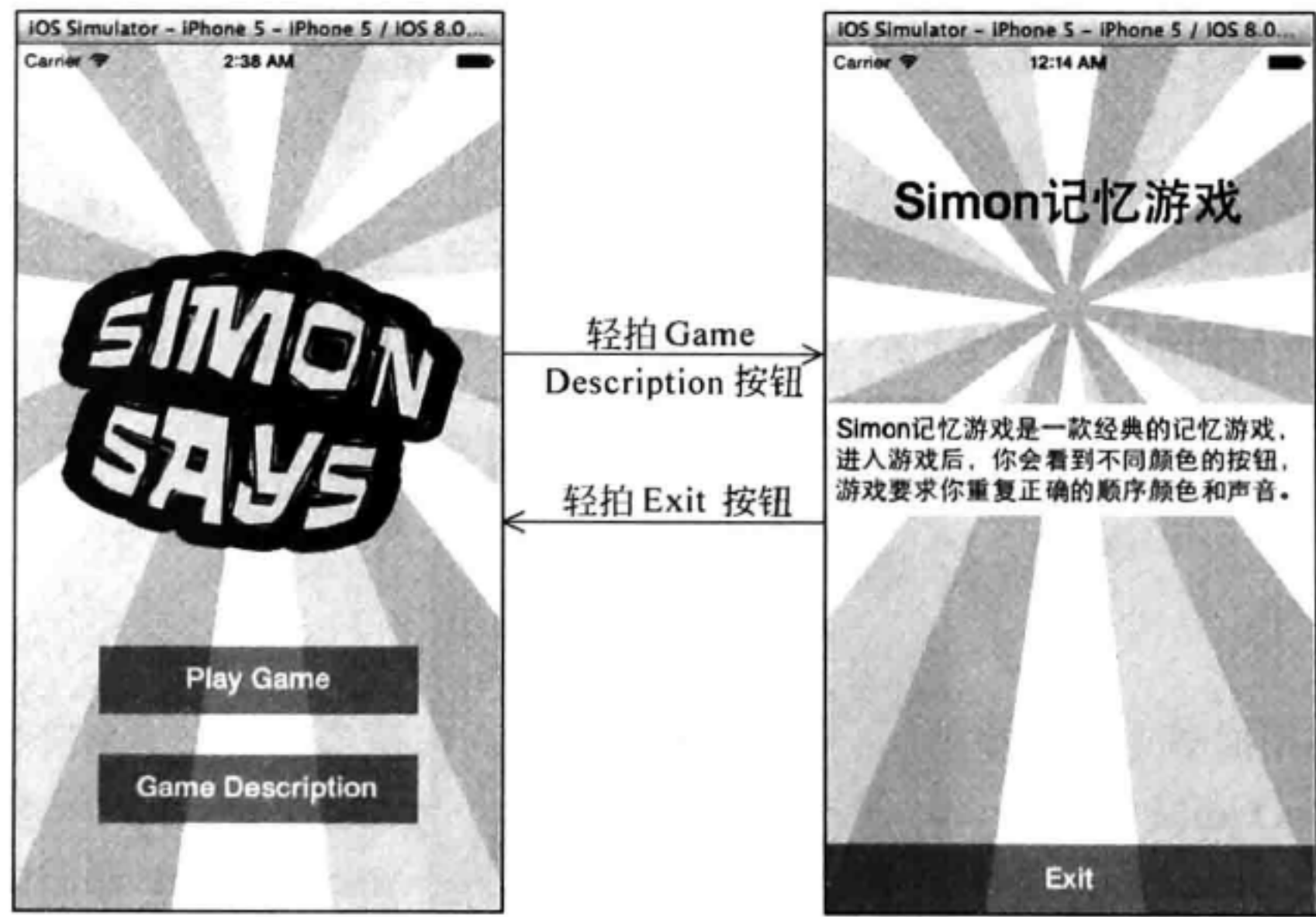


图 8.23 运行效果 2

第9章 迷你高尔夫——用户交互

迷你高尔夫游戏是由高尔夫演变而来的。只要个人或团体球员以不同的高尔夫球杆将一颗高尔夫球打进洞内就可以了。由于高尔夫这项体育运动的要求比较特殊，例如场地、服饰、球杆等，其花费是十分不菲的，所以对于屌丝来说只是一项可望而不可及的运动。本章将为广大屌丝完成高尔夫的梦想，实现一款迷你高尔夫游戏，让玩家可以在任何地方、任何时间开始高尔夫的比赛。在本章中将着重讲解游戏中的用户交互问题，如玩家操控高尔夫球，以及游戏对高尔夫球的各种约束。

9.1 游戏介绍

迷你高尔夫是一款体育方面的游戏。玩家可以通过以下两种方式进行操作：

- ❑ 触摸点击高尔夫球，拖动调整角度和发射的力度，松开高尔夫球弹出。
- ❑ 直接触摸屏幕的某一地方使高尔夫球弹出。

当高尔夫球进入洞中就胜利了。这样的游戏，通常包括以下几个模块。

1. 主菜单模块

在主菜单的界面上提供了两个菜单项，如图 9.1 所示。轻拍 Play Game 菜单项，进入游戏界面；轻拍 Game Description 菜单项，进入游戏介绍界面。

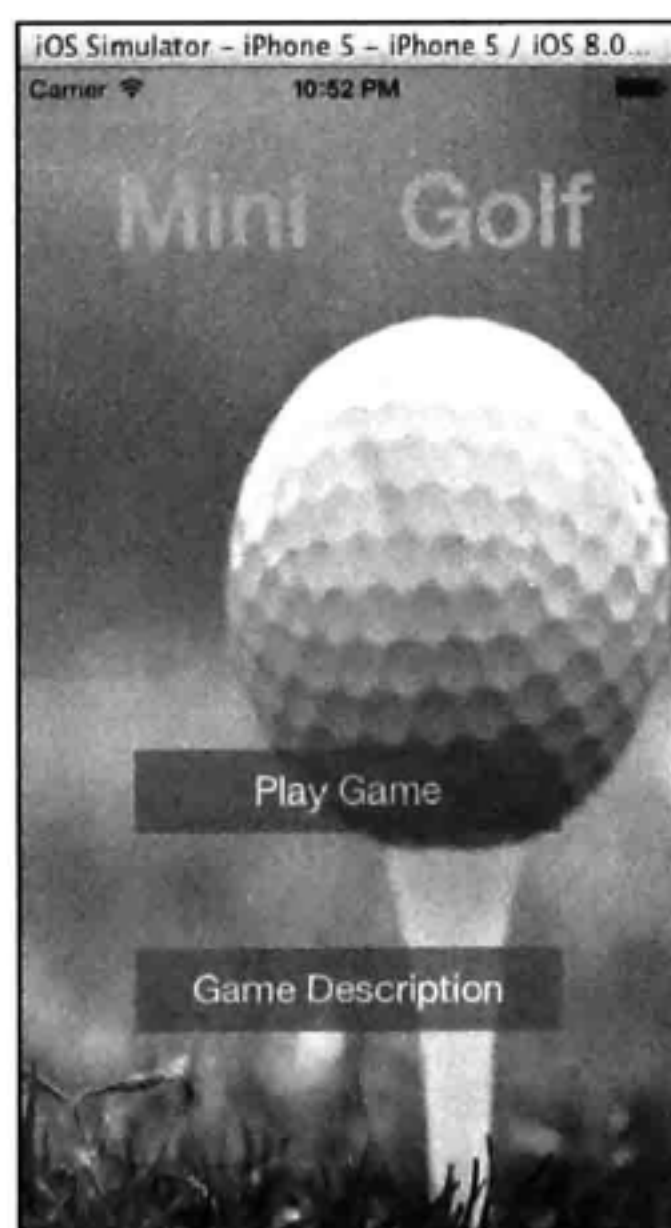


图 9.1 主菜单模块

2. 游戏模块

游戏模块提供了游戏的界面，如图 9.2 所示。玩家可以在此界面中进行一些操作，并且做出相应的响应，即实现将球打入洞中。

3. 游戏介绍模块

游戏介绍模块提供了游戏玩法的说明，如图 9.3 所示。

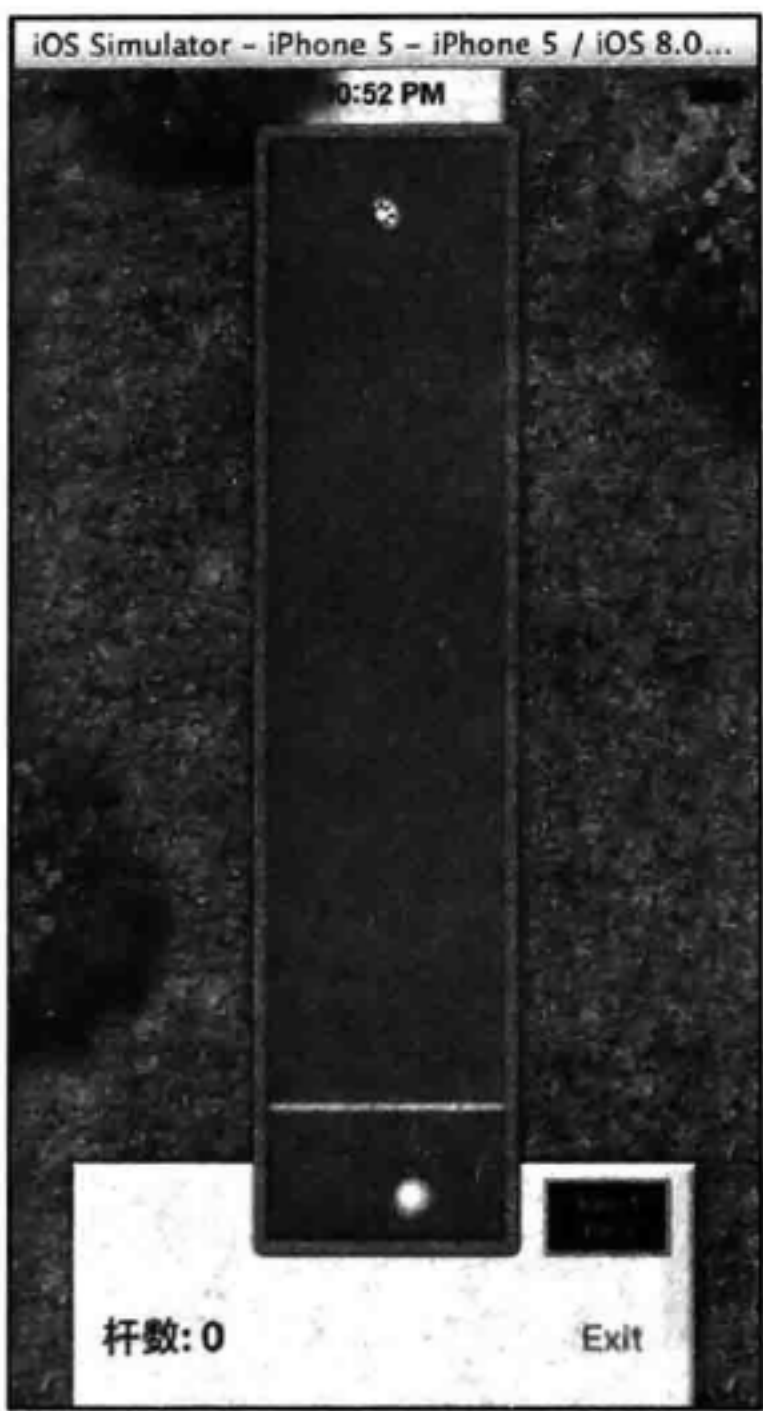


图 9.2 游戏模块

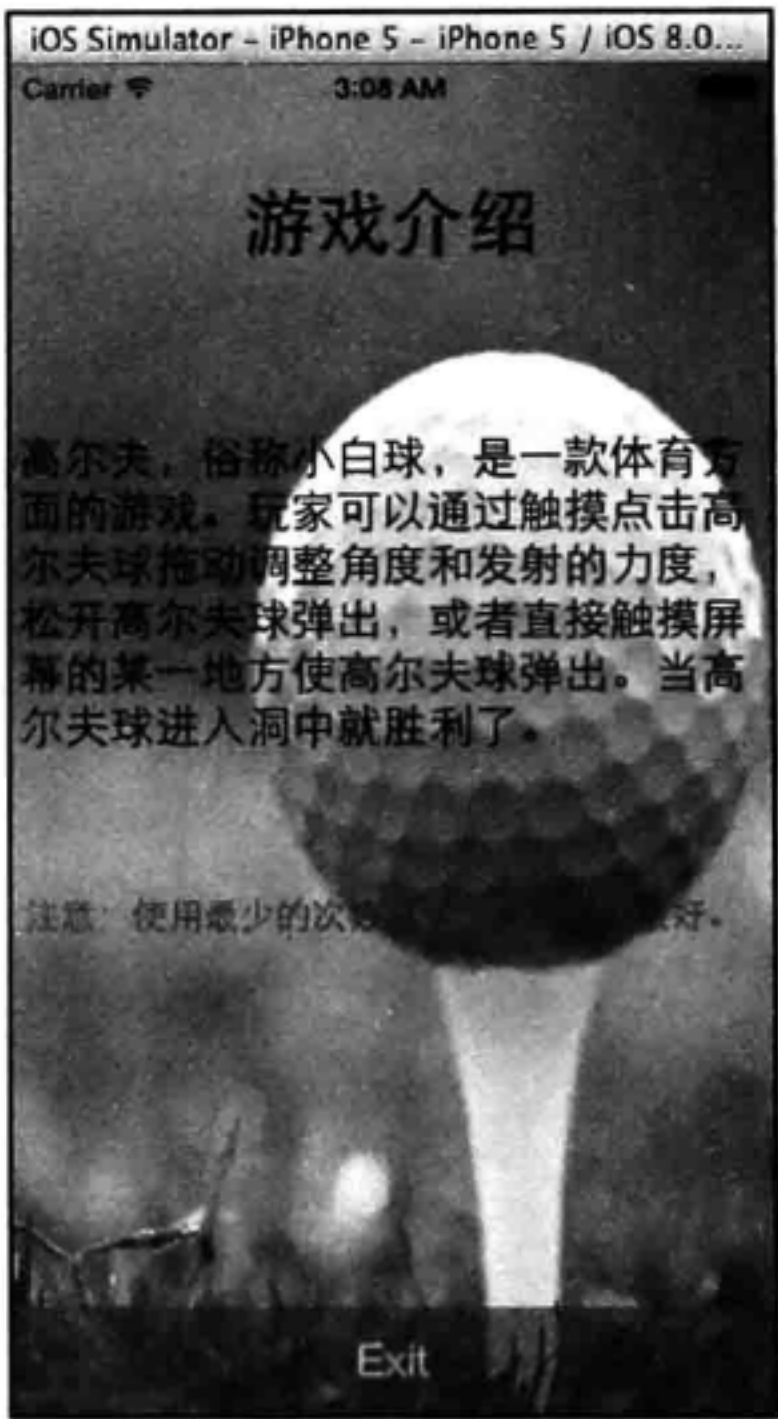


图 9.3 游戏介绍模块

9.2 开发游戏之前的准备工作

在开发迷你高尔夫游戏之前，需要做一些准备工作，这些准备工作如下所述。

- (1) 创建一个 Single View Application 模板类型的项目，命名为 Mini Golf。
- (2) 添加图像 Background、ball.png、cup.png 和 gameBackground.png 到创建项目的 Supporting Files 文件夹中，如图 9.4 所示。



图 9.4 添加的图像

9.3 主菜单模块

在迷你高尔夫中也有一个主菜单，主菜单中的菜单项也是由按钮实现的。本节将讲解主菜单的设计。

双击将 Main.storyboard 文件打开，对主菜单的界面进行设计，具体的操作步骤如下所述。

(1) 选择 Show the File inspector 选项，将 Interface Builder Document 中的 Opens in 改为 Xcode 5.1。

(2) 对在画板中原本存在的 View Controller 视图控制器的界面进行设计，效果如图 9.5 所示。

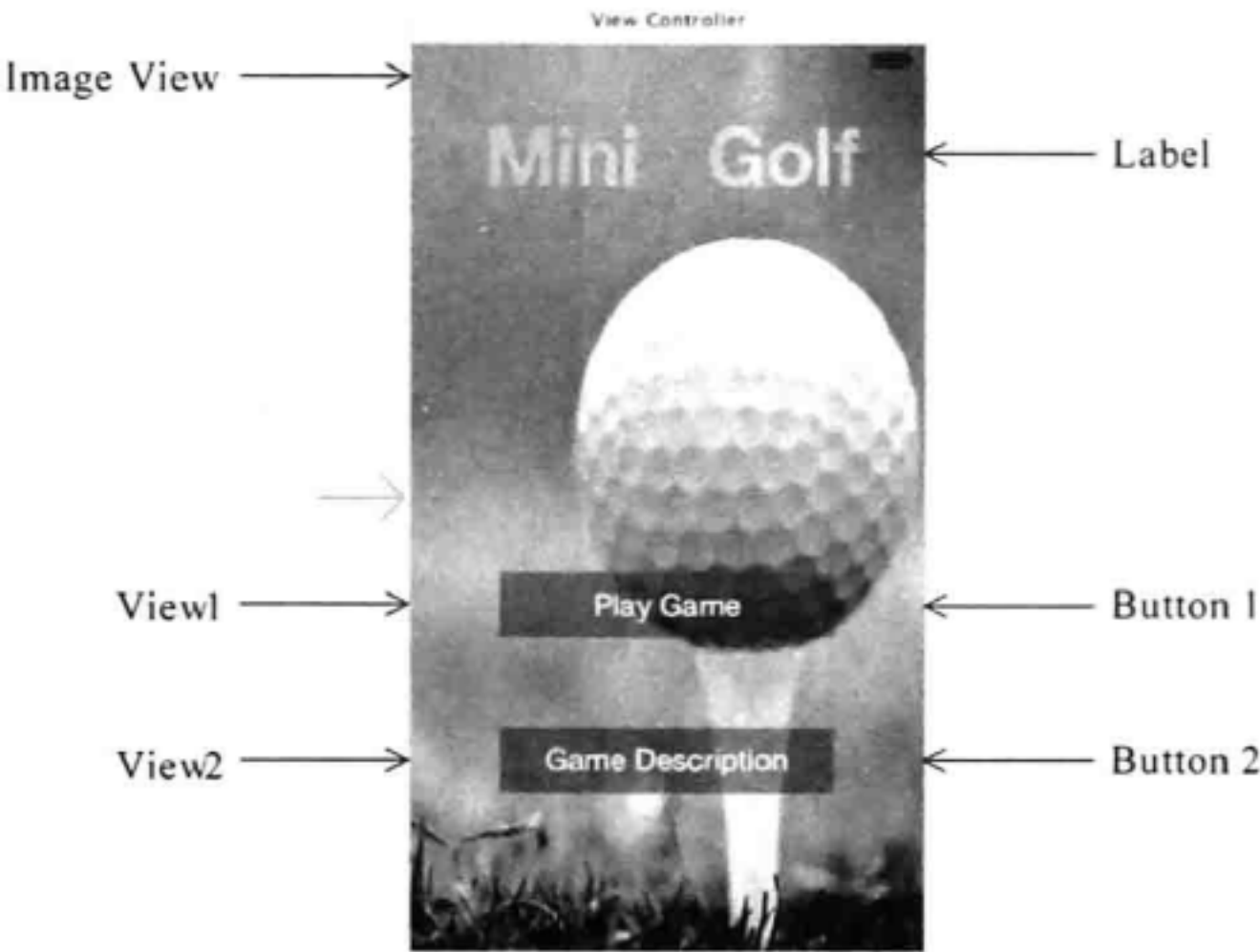


图 9.5 界面的效果

需要添加的视图对象，以及对它们的设置，如表 9-1 所示。

表 9-1 设置界面

视 图	设 置
Image View	Image: Background.png 位置和大小: (0, 0, 320, 568)
Label	Text: Mini Golf Color: 绿色 Font: System Bold 50.0 Alignment: 居中 位置和大小: (19, 42, 283, 56)
Button1	Title: Play Game Font: System 19 Text Color: 白色 位置和大小: (35, 6, 138, 30)
Button2	Title: Game Description Font: System 19 Text Color: 白色 位置和大小: (25, 6, 159, 30)

续表

视 图	设 置
View1	Alpha: 0.6 Background: 深灰色 位置和大小: (56, 331, 208, 41)
View2	Alpha: 0.6 Background: 深灰色 位置和大小: (56, 428, 208, 41)

9.4 游 戏 模 块

游戏模块是一个游戏中很重要的部分。在该模块中，实现玩家对高尔夫球的控制。本节将讲解有关游戏模块的界面设计、高尔夫球的添加，以及高尔夫球的移动等内容。

9.4.1 准备工作

在设计游戏界面前，需要做一些准备工作。例如为了便于代码的管理，将每一个界面实现的功能代码保存在一个文件中。

1.创建文件

在创建的项目中需要创建一个 Swift File 模板类型的文件，命名为 MiniGolfViewController。

2.创建空类

打开创建的 MiniGolfViewController.swift 文件，在此文件中创建一个基于 UIViewController 类的空类 SimonGameViewController，代码如下：

```
import UIKit
class MiniGolfViewController: UIViewController {
}

```

该类用于编写高尔夫游戏的代码。

9.4.2 界面设计

以下是对游戏界面进行设计的具体操作步骤。

(1) 在视图对象库中拖动 View Controller 视图控制器对象到画布中。

(2) 单击新添加的视图控制器，选择界面上方的 Dock 中的 View Controller 图标。在工具窗口中的 Show the Identity inspector 选项，即标示查看器中，将 Custom Class 下的 Class 设置为创建的 MiniGolfViewController 类。这时在画布中的这个视图控制器就变为了 Mini Golf View Controller 视图控制器。

(3) 对 Mini Golf View Controller 视图控制器的界面进行设计，效果如图 9.6 所示。

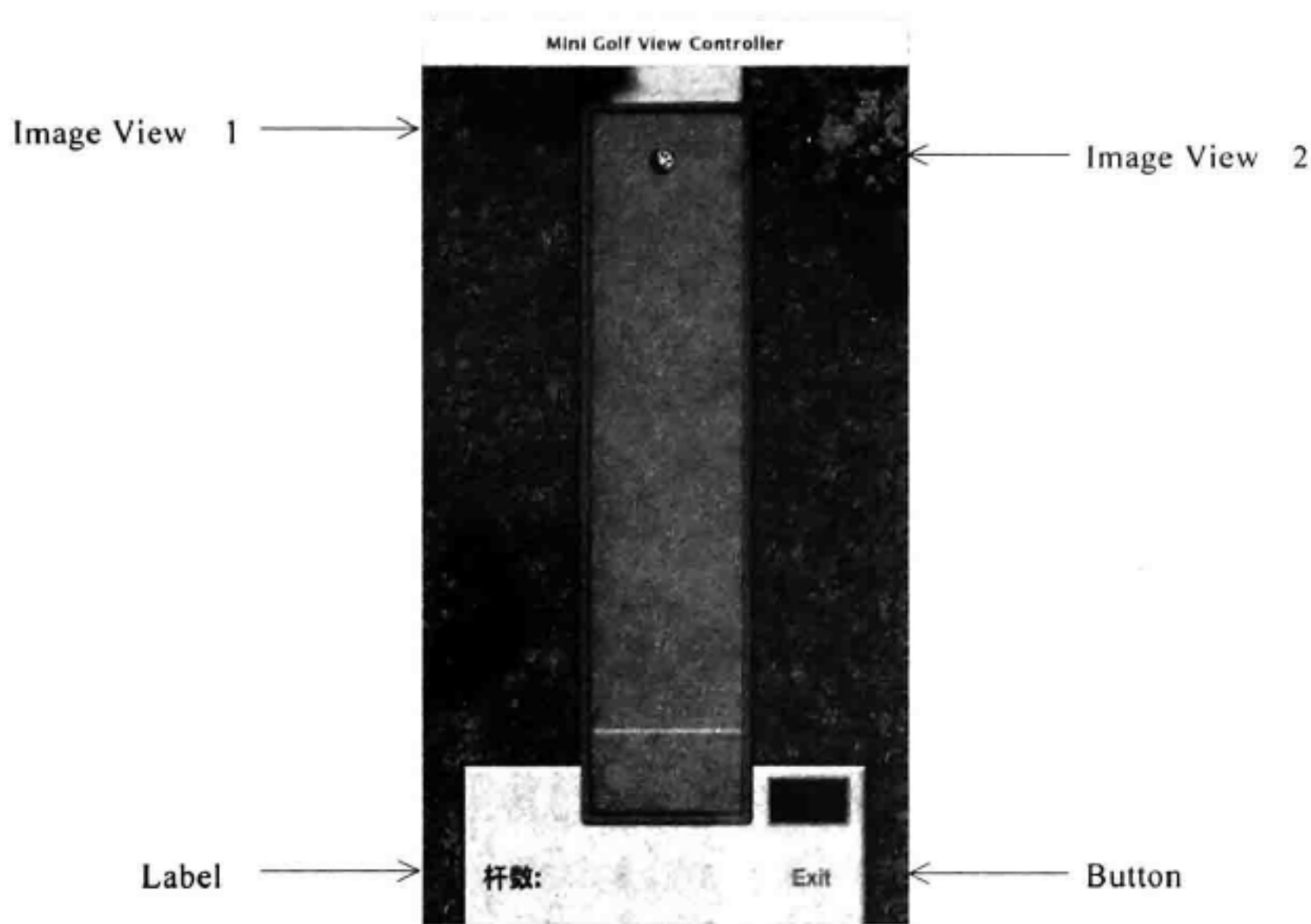


图 9.6 界面效果

需要添加的视图对象，以及对它们的设置，如表 9-2 所示。

表 9-2 设置界面

视 图	设 置
Image View1	Image: gameBackground.png 位置和大小: (0, 0, 320, 568)
Image View2	Image: cup.png 位置和大小: (148, 50, 24, 24)
Label	Text: 杆数: Font: System Bold 17.0 位置和大小: (39, 530, 141, 21) 与 MiniGolfViewController.swift 文件声明并关联一个插座变量 StrokeCount
Button	Title: Exit Font: System Bold 19.0 位置和大小: (232, 530, 46, 21)

9.4.3 添加高尔夫球

要实现高尔夫的游戏，首先需要在游戏的界面添加一个高尔夫球，高尔夫球的添加是使用图像视图实现的。以下是添加高尔夫的具体步骤。

(1) 打开 MiniGolfViewController.swift 文件，在此文件中声明 4 个变量。代码如下：

```
var gameState:NSString?=nil
var playerBall:UIImage=UIImage(named: "ball.png")
var playerBallView:UIImageView=UIImageView()
var ballRect:CGRect=CGRect()
```

其中，gameState 是用来保存游戏状态的，playerBallView 是图像视图对象，用来显示高尔夫球。

(2) 添加 viewDidLoad()方法, 在此方法中实现高尔夫球的显示, 代码如下:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.gameState="Loading"
    self.playerBallView.image=self.playerBall //设置图像视图显示的图像
    var screenheight:CGFloat=UIScreen mainScreen().bounds.size.height //获取屏幕的高度
    self.ballRect=CGRectMake(160, screenheight - 100, 24, 24)
    self.playerBallView.frame=self.ballRect //设置图像视图的框架
    self.view.addSubview(self.playerBallView) //将图像视图添加对主视图中
}
```

(3) 回到 Main.storyboard 文件中。单击 Mini Golf View Controller 视图控制器。在此视图控制器中, 选择界面上方的 Dock 中的 Mini Golf View Controller 图标。在工具窗口中的 Show the Attributes inspector 选项, 即属性查看器中, 找到 Is Initial View Controller 复选框, 将其选中。此时 Mini Golf View Controller 视图控制器就成为了初始视图控制器。此时运行程序, 会看到如图 9.7 所示的效果。

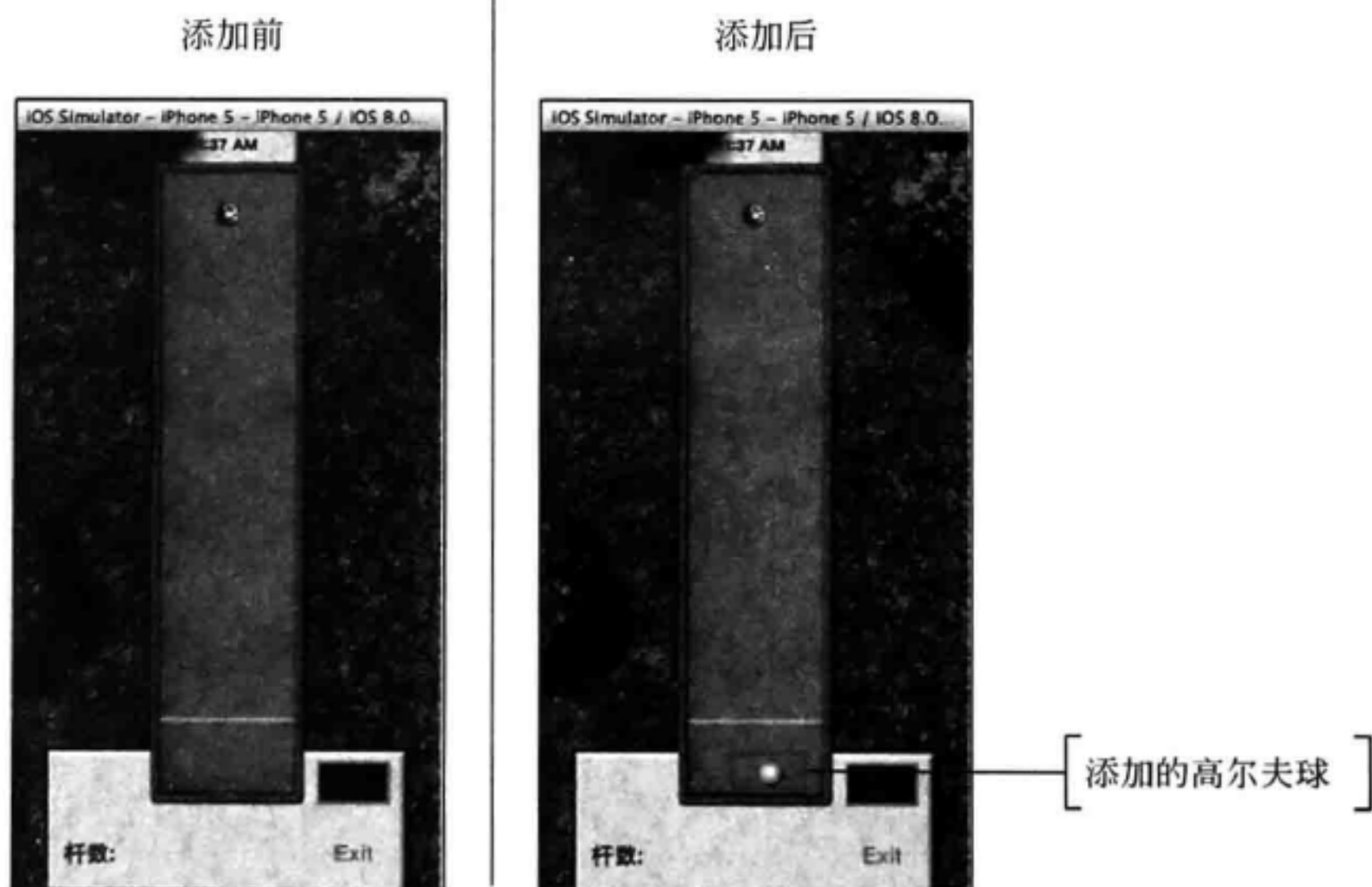


图 9.7 运行效果

9.4.4 移动高尔夫球

在正规的高尔夫球运动中, 高尔夫球是被人用高尔夫球杆击打一次或多次, 然后进入目标洞的过程, 这样就完成了一次完美的人与高尔夫球的交互运动。在我们的游戏中, 玩家与游戏的交互运动是不需要使用球杆的, 而是通过触摸事件让高尔夫球实现移动, 也就是现实中的击打效果。当玩家在屏幕上触摸结束后, 将会获取一个位置, 这个位置可以用来更新高尔夫球的位置, 又由于定时器的关系, 可以使高尔夫球的移动形成动画效果。这里的触摸 (Cocoa Touch) 是指, 当玩家的手指放在屏幕上进行操作并一直到手指离开的全过程。触摸是在 UIView 上进行的。当玩家触摸到屏幕时, 触摸事件就会发生。触摸事件如表 9-3 所示。

表 9-3 触摸事件

方 法	功 能
touchesBegan(touches: NSSet, withEvent event: UIEvent)	手指刚接触到屏幕
touchesMoved(touches: NSSet, withEvent event: UIEvent)	手指在屏幕上移动
touchesEnded(touches: NSSet, withEvent event: UIEvent)	结束触摸
touchesCancelled(touches: NSSet!, withEvent event: UIEvent!)	取消触摸

以下就是通过触摸手势实现高尔夫球移动的具体步骤。

(1) 需要声明 5 个变量，代码如下：

```
var updateTimer:NSTimer?=nil           //此定时器对象的动作是用来实现动画效果的
var ballDirection:CGPoint=CGPointMake(0, 0)           //保存球的轨迹
var ballVelocity:Float=0                 //保存球的力度
var touchStartPos:CGPoint=CGPoint()       //保存玩家开始触摸的位置
var touchEndPos:CGPoint=CGPoint()        //保存结束触摸的位置
```

(2) 在 viewDidLoad()方法中添加以下的代码，让定时器可以在每隔 0.03 秒就调用一次 update()方法：

```
self.updateTimer=NSTimer.scheduledTimerWithTimeInterval(0.03, target:
self, selector: Selector("update"), userInfo: nil, repeats: true)
//创建定时器
self.gameState="Waiting"
```

(3) 实现触摸事件。在 touchesBegan(touches: NSSet, withEvent event: UIEvent)方法中编写代码，实现手指在屏幕上触摸的位置。

```
override func touchesBegan(touches: NSSet, withEvent event: UIEvent) {
//判断当前游戏的状态是否为 Playing
if((self.gameState?.isEqualToString("Playing")) == true){
return
}
var touch: AnyObject=touches.anyObject()!
touchStartPos=touch.locationInView(touch.view)           //获取触摸的位置
}
```

(4) 在 touchesMoved(touches: NSSet, withEvent event: UIEvent)方法中编写代码，实现手指在屏幕上移动从而获取当前的触摸位置。代码如下：

```
override func touchesMoved(touches: NSSet, withEvent event: UIEvent) {
var touch: AnyObject=touches.anyObject()!
touchEndPos=touch.locationInView(touch.view)
}
```

(5) 在 touchesEnded(touches: NSSet, withEvent event: UIEvent)方法中编写代码，实现对距离的获取，代码如下：

```
override func touchesEnded(touches: NSSet, withEvent event: UIEvent) {
//判断当前游戏的状态是否为 Playing
if((self.gameState?.isEqualToString("Playing")) == true){
return
}
```



```

}
var distance=sqrt(pow((touchEndPos.x-touchStartPos.x),
2.0)+pow((touchEndPos.y-touchStartPos.y), 2.0)) //获取两点的距离
//判断距离是否大于0
if(distance>0){
    self.gameState="Playing"
    ballVelocity = Float(distance) * 0.01
    ballDirection = CGPointMake((touchEndPos.x - touchStartPos.x) * 0.01,
(touchEndPos.y - touchStartPos.y) * 0.01)
}
}

```

(6) 在 touchesCancelled(touches: NSSet!, withEvent event: UIEvent!) 方法中编写代码, 实现对 touchStartPos 和 touchEndPos 等对象的重新设置。代码如下:

```

override func touchesCancelled(touches: NSSet!, withEvent event: UIEvent!)
{
    touchStartPos = CGPointMake(0,0)
    touchEndPos = CGPointMake(0,0)
    ballVelocity = 0
    self.gameState = "Waiting"
}

```

(7) 添加 update() 方法, 在此方法中实现高尔夫球的移动, 代码如下:

```

func update(){
    //判断当前游戏的状态是否为 Playing
    if((self.gameState?.isEqualToString("Playing")) == false){
        return
    }
    self.ballRect=CGRectOffset(self.ballRect, ballDirection.x*CGFloat
(ballVelocity), ballDirection.y*CGFloat(ballVelocity))
    self.playerBallView.frame=self.ballRect //设置图像视图的位置
}

```

此时运行程序, 会看到如图 9.8 所示的效果。

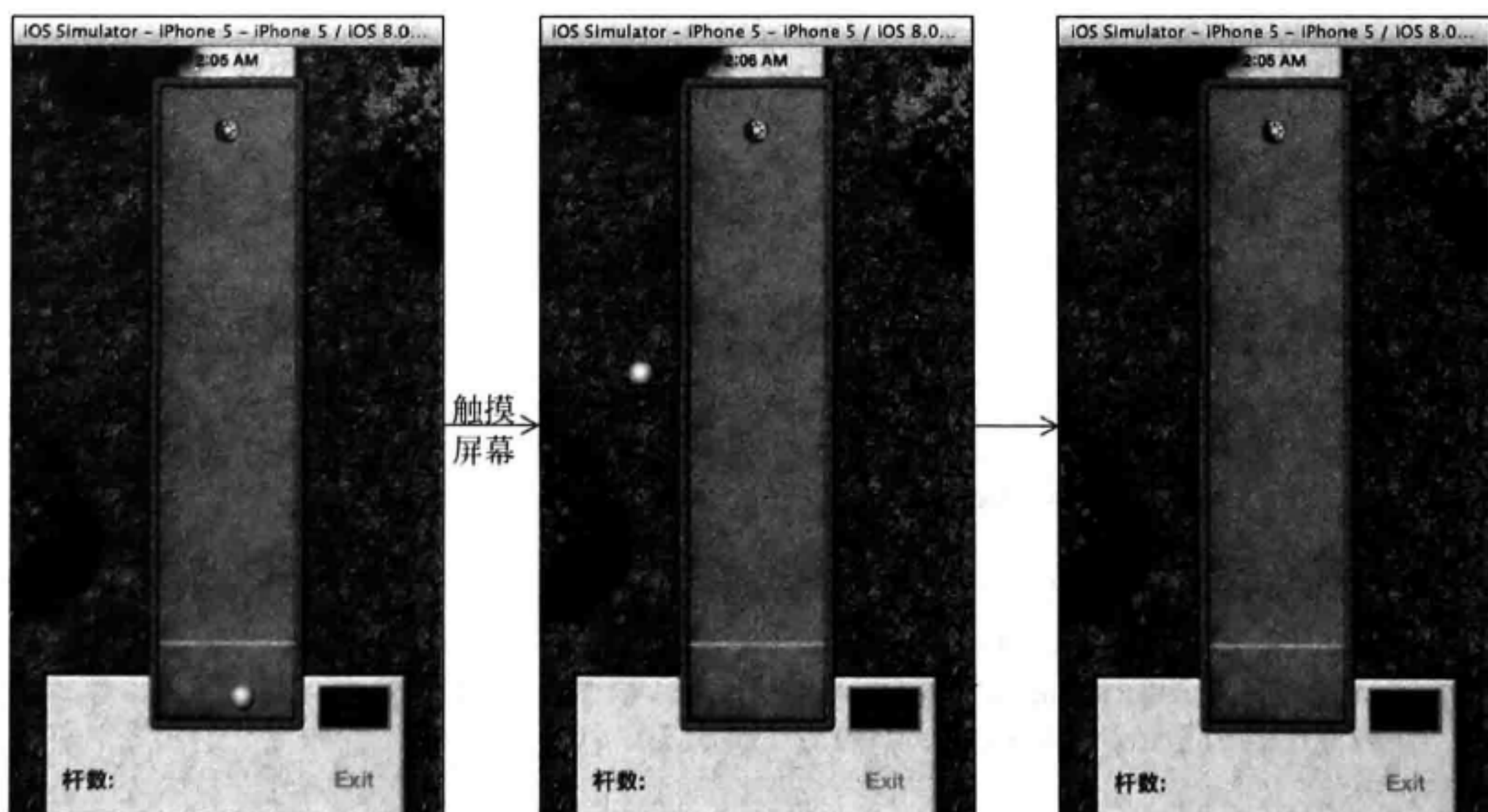


图 9.8 运行效果

9.5 用户交互中的不足

在 9.4.4 节中, 可以看到通过玩家触摸后高尔夫球确实实现了移动, 但是此时的游戏还存在着一些问题, 例如, 高尔夫球无边界的滚动, 以及没有球进入洞中的限制等。本节将弥补这些在用户交互中存在的不足。

9.5.1 边界的限定

在图 9.8 中不难看出, 高尔夫球的在移动时, 会超出屏幕范围, 此时需要为高尔夫球的运动范围添加一个边界, 使它只可以在此边界范围内移动。实现方式如下所述。

(1) 添加边界的限定首先需要实例化一个 `courseBounds` 对象, 此对象用来当做一个限制边界, 代码如下:

```
var courseBounds:CGRect=CGRect()
```

(2) 在 `viewDidLoad()`方法中添加以下代码, 为这个对象设置位置和大小:

```
courseBounds=CGRectMake(110,30, 100,470)
self.updateTimer=NSTimer.scheduledTimerWithTimeInterval(0.03, target:
self, selector: Selector("update"), userInfo: nil, repeats: true)
```

(3) 在 `update()`方法中添加以下代码, 实现对高尔夫球是否超出边界的判断:

```
if((self.gameState?.isEqualToString("Playing")) == false){
    return
}
//判断高尔夫球的 x 的位置是否超出了 courseBounds 即边界的 x 的位置
if(self.ballRect.origin.x <= courseBounds.origin.x){
    ballDirection.x=fabs(ballDirection.x)
}
//判断高尔夫球的 x 的位置加上宽度是否大于边界的 x 的位置加上边界的宽度
else if ((self.ballRect.origin.x + self.ballRect.size.width) >=
(courseBounds.origin.x+courseBounds.size.width)){
    ballDirection.x = -fabs(ballDirection.x)
}
//判断高尔夫球的 y 的位置是否超出了 courseBounds 即边界的 y 的位置
if (self.ballRect.origin.y <= courseBounds.origin.y) {
    ballDirection.y = fabs(ballDirection.y)
}
//判断高尔夫球的 x 的位置加上宽度是否大于边界的 x 的位置加上边界的宽度
else if (self.ballRect.origin.y + self.ballRect.size.height >=
(courseBounds.size.height+courseBounds.origin.y)){
    ballDirection.y = -fabs(ballDirection.y)
}
```

此时运行程序, 会看到如图 9.9 所示的效果。

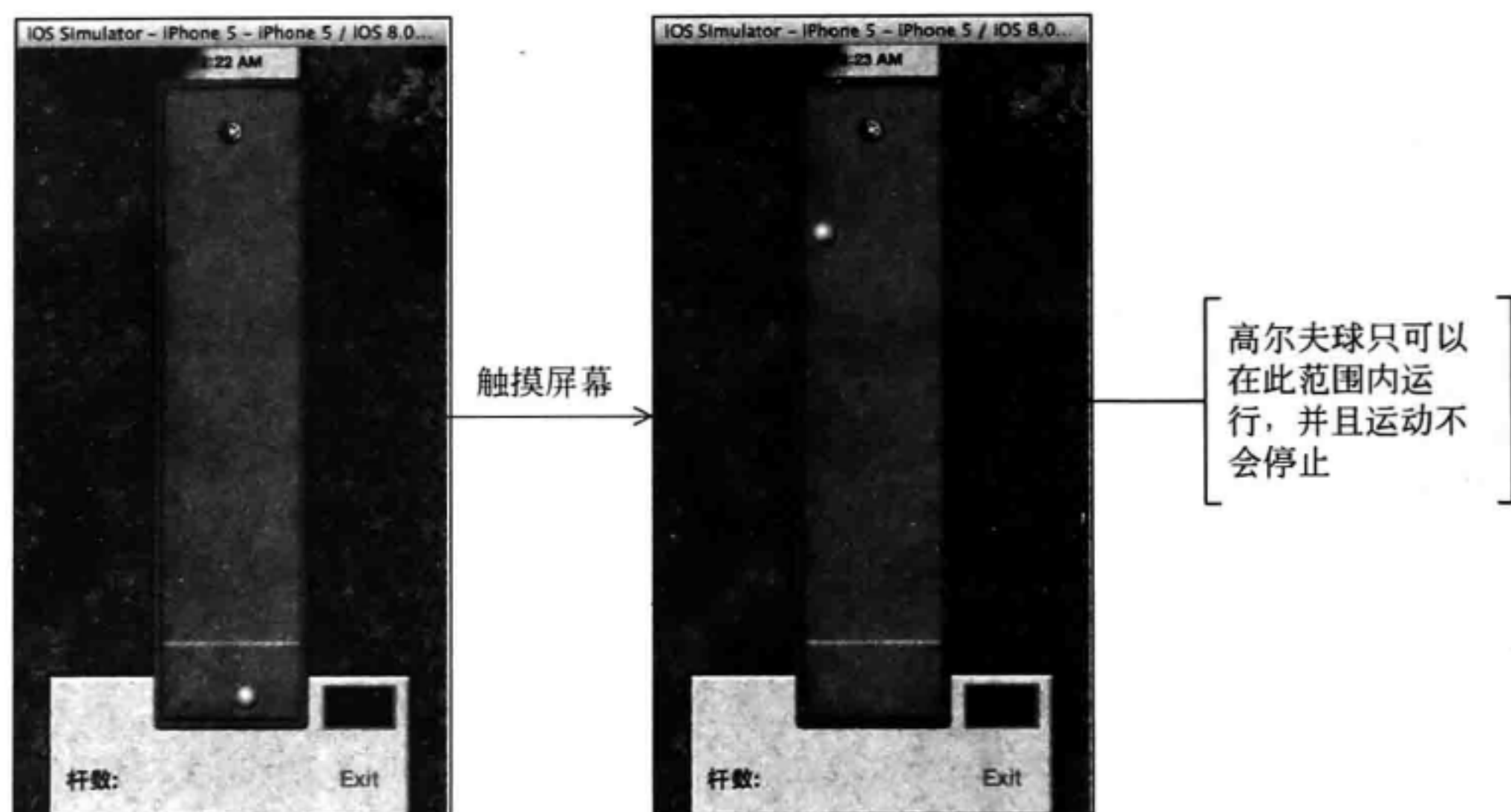


图 9.9 运行效果

9.5.2 速度限定

在 9.5.1 节中，如果是高尔夫球移动了，它就会无休止的移动下去。如何使高尔夫球停止就是本小节将要讲解的内容。要使移动的高尔夫球停止下来，我们需要使用到 `ballVelocity` 变量，此变量用来保存高尔夫球的速度，将此变量中保存的值慢慢变为 0 就可以了，即使高尔夫球做减速运动。在 `update()` 方法中添加以下的代码：

```
else if (self.ballRect.origin.y + self.ballRect.size.height >=
(courseBounds.size.height+courseBounds.origin.y)){
    ballDirection.y = -fabs(ballDirection.y)
}
//判断速度是大于 0
if(ballVelocity>0){
    //如果大于 0 就将 ballVelocity 减 0.05，即让 ballVelocity 减速
    ballVelocity-=0.05
}
//判断速度是小于等于 0
if(ballVelocity <= 0){
    self.gameState = "Waiting"
    ballVelocity = 0
}
```

此时运行程序，会看到如图 9.10 所示的效果。

9.5.3 进洞的限定

高尔夫这一项运行在高尔夫球进入球洞后才算赢，如何判断高尔夫是否进入洞中是本小节将要讲解的内容。判断高尔夫是否进入洞中需要利用到 `CGRectIntersectsRect`

(CGRect, CGRect)方法, 以下是它的具体实现。

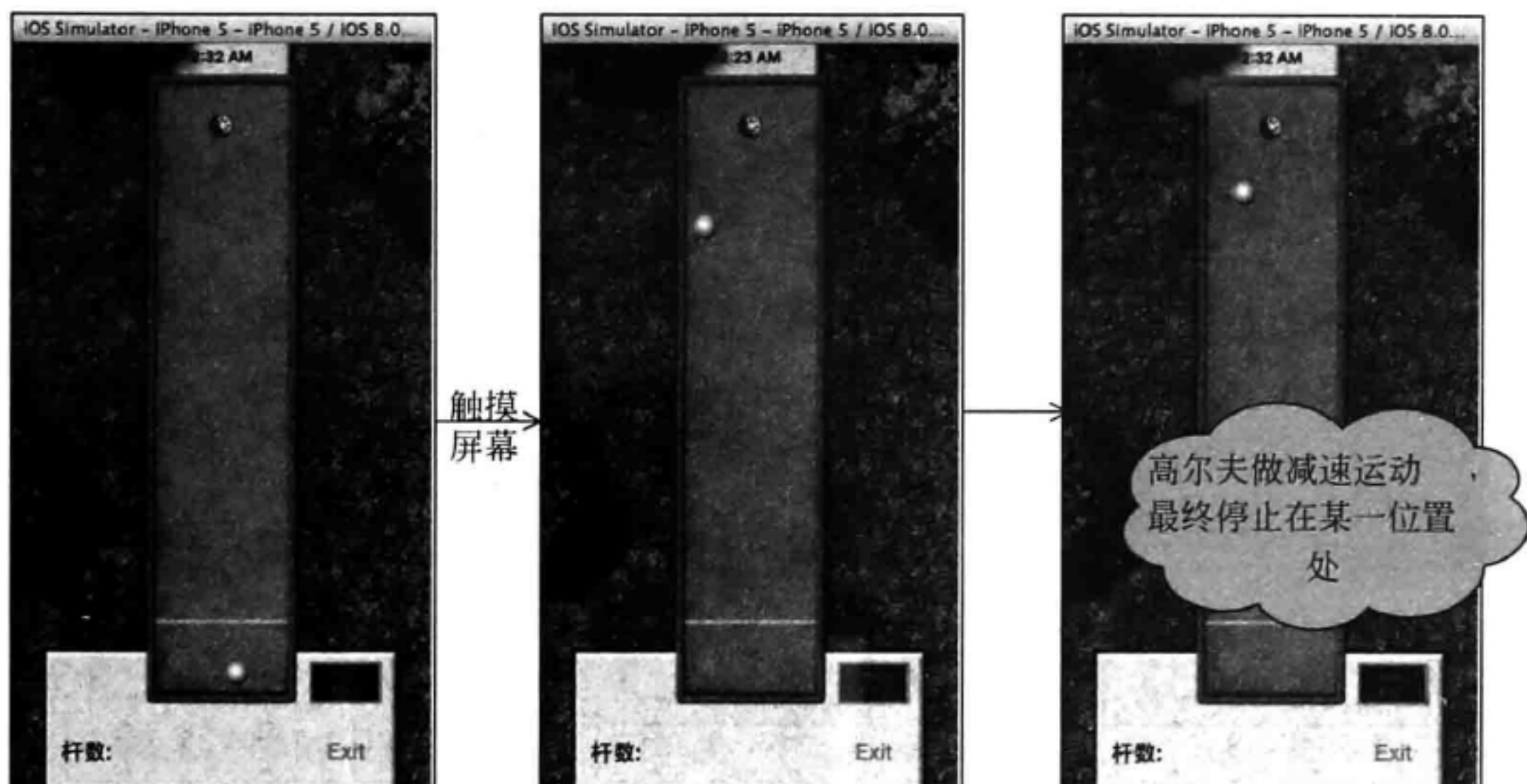


图 9.10 运行效果

(1) 需要实例化一个对象 cupRect, 代码如下:

```
var cupRect:CGRect=CGRect()
```

(2) 需要在 viewDidLoad()方法中为实例化的 cupRect 对象赋值, 代码如下:

```
self.cupRect=CGRectMake(152, 57, 10,10)
courseBounds=CGRectMake(110,30, 100,470)
```

(3) 需要在 update()方法中使用 CGRectIntersectsRect(_rect1:CGRect, _rect2:CGRect)方法进行判断, 判断高尔夫球的矩形对象是否和 cupRect 的矩形框架重叠, 如果重叠表明高尔夫球已经入洞中, 如果不重叠表明高尔夫球没有进入洞中, 代码如下:

```
if (CGRectIntersectsRect(self.ballRect, self.cupRect)) {
    //如果重叠
    self.gameState = "Done"
    self.ballRect = CGRectMake(150, 55, 12, 12);
    ballVelocity = 0
    //对警告视图的设置
    var alert:UIAlertView=UIAlertView()
    alert.title="球终于进了"
    alert.message="是否重新进入游戏"
    alert.addButtonWithTitle("是")
    alert.addButtonWithTitle("否")
    alert.show()
    alert.delegate=self
}
self.ballRect=CGRectOffset(self.ballRect,
ballDirection.x*CGFloat(ballVelocity),
ballDirection.y*CGFloat(ballVelocity))
```

此时运行程序, 会看到如图 9.11 所示的效果。

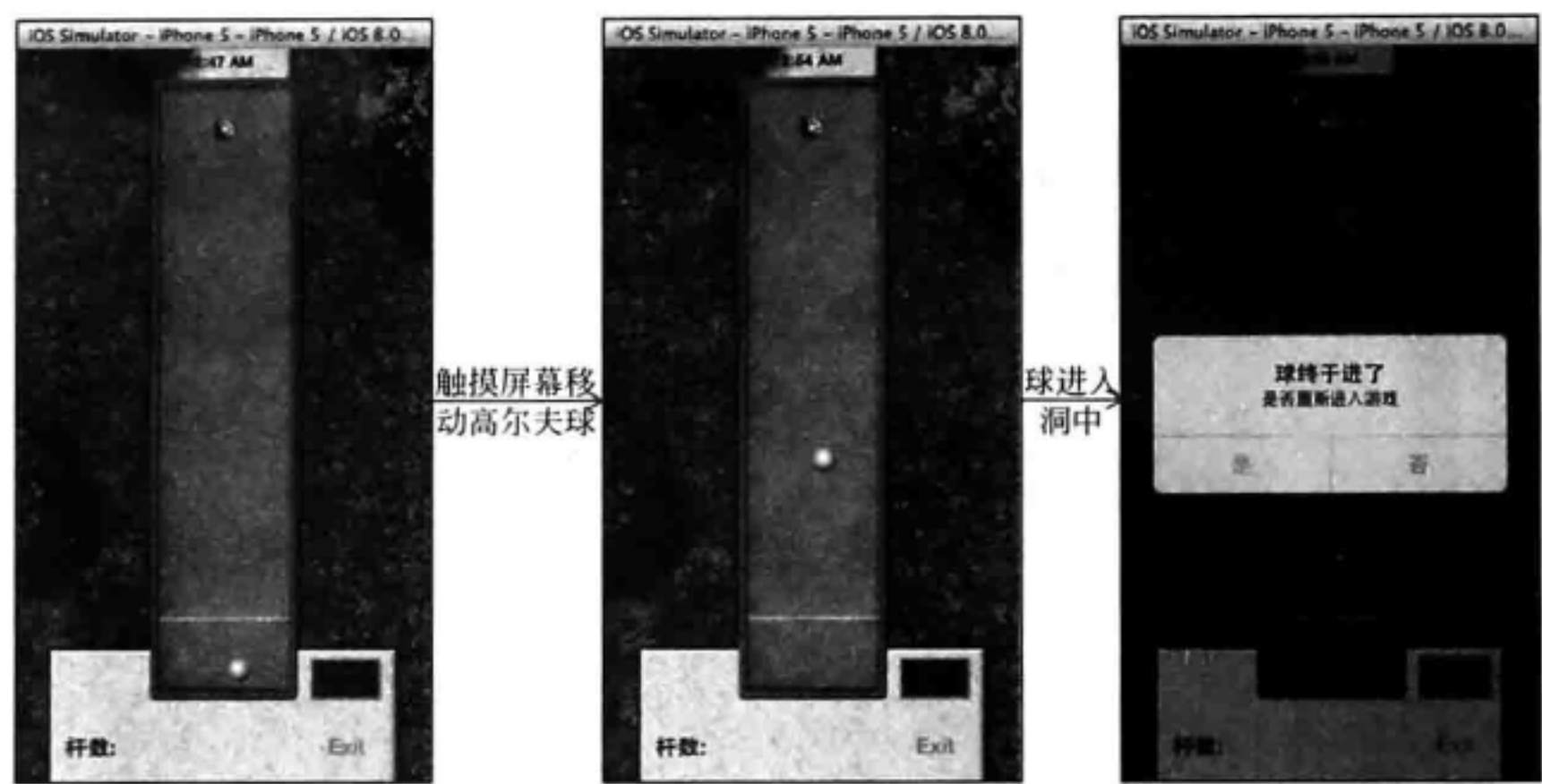


图 9.11 运行效果

在图 9.11 中可以看到，当高尔夫球进入球洞后会弹出一个“球终于进了”的警告视图，在此警告视图中出现了两个按钮，以下就是实现这两个按钮响应的代码：

```
func alertView(_alertView: UIAlertView,clickedButtonAtIndex buttonIndex:
Int){
    var name:NSString=_alertView.buttonTitleAtIndex(buttonIndex)
    //判断当前按下的按钮是否为"是"按钮
    if(name.isEqualToString("是")){
        //如果是"是"按钮，就重新开始游戏，将显示高尔夫球的图像视图的框架改变为最初的框架即
        (160, screenheight - 100, 24, 24)
        var screenheight:CGFloat=UIScreen.mainScreen().bounds.size.height
        self.ballRect=CGRectMake(160, screenheight - 100, 24, 24)
        self.playerBallView.frame=self.ballRect
    }
    else{
        //如果是"否"按钮，就退出游戏
        exit(1)
    }
}
```

此时运行程序，会看到以下的效果。当触摸警告视图中的“是”按钮时，就重新开始游戏，将显示高尔夫球的图像视图的框架改变为最初的框架，如图 9.12 所示。

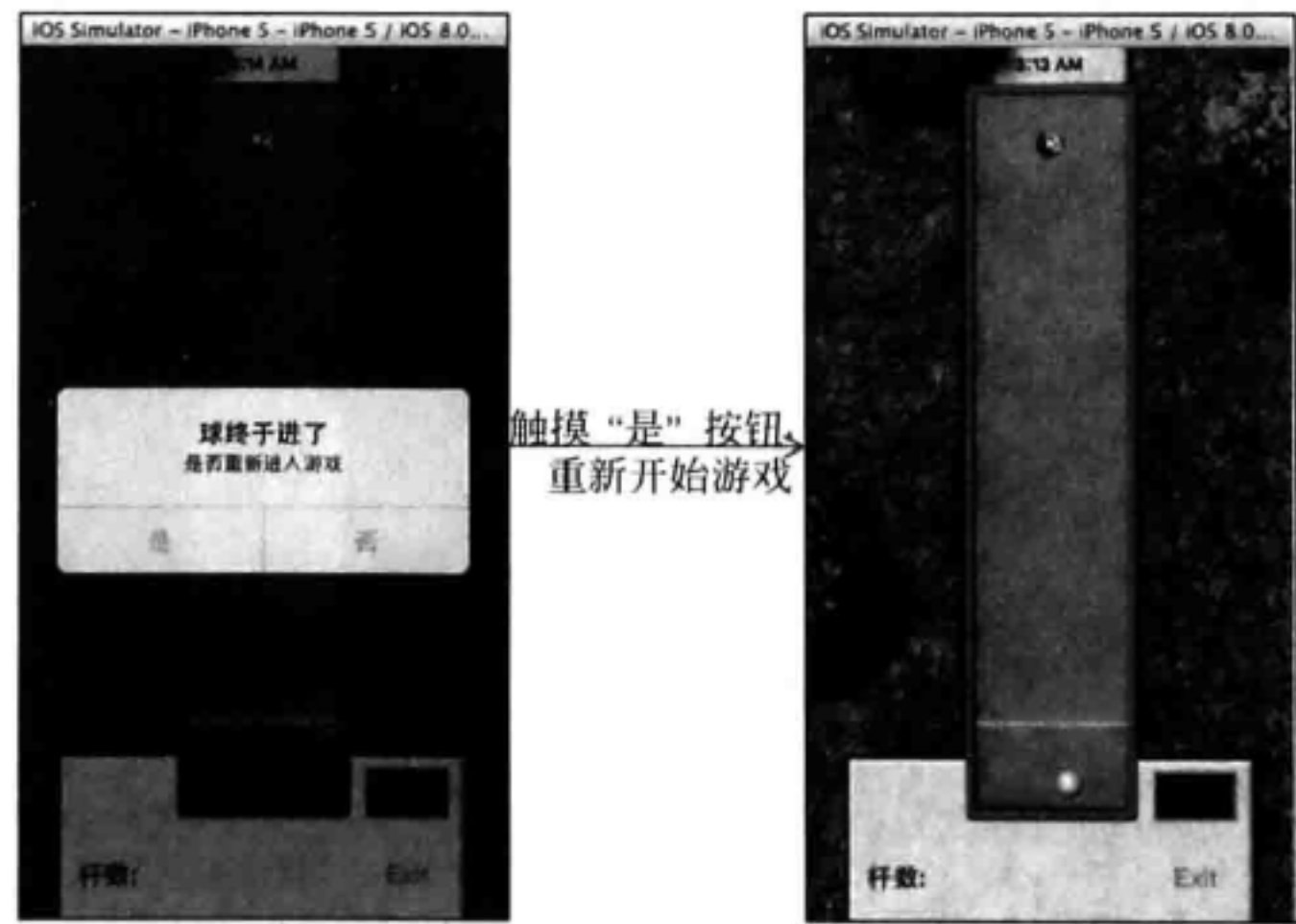


图 9.12 运行效果 1

如果触摸警告视图中的“否”按钮，就退出游戏，如图 9.13 所示。



图 9.13 运行效果 2

9.6 杆数显示

本节将讲解杆数的显示，这样一来可以让玩家轻松地看到高尔夫球从开始一直到进入洞中所需要的杆数。需要注意的是，显示的杆数越少越好。

(1) 需要声明一个变量，此变量用来保存玩家对球进行操作的次数，代码如下：

```
var strokeCounter:Int=0
```

(2) 在 viewDidLoad()方法中添加以下的代码，实现将 strokeCounter 变量中保存的值显示在标签中：

```
courseBounds=CGRectMake(110,30, 100,470)
self.StrokeCount.text="杆数: \(strokeCounter) " //在标签中显示内容
```

(3) 在 touchesEnded(touches: NSSet, withEvent event: UIEvent)方法中编写以下的代码，实现当玩家每触摸结束后 strokeCounter 变量中保存值自动加 1，并将值显示在标签中：

```
if(distance>0){
    self.gameState="Playing"
    strokeCounter++
    self.StrokeCount.text="杆数: \(strokeCounter) "
    ballVelocity = Float(distance) * 0.01
    ballDirection = CGPointMake((touchEndPos.x - touchStartPos.x) * 0.01,
    (touchEndPos.y - touchStartPos.y) * 0.01)
}
```

(4) 在 alertView(_alertView: UIAlertView,clickedButtonAtIndex buttonIndex: Int)方法中编写以下代码：


```
if(name.isEqualToString("是")){
    ar screenHeight:CGFloat=UIScreen mainScreen().bounds.size.height
    elf.ballRect=CGRectMake(160, screenHeight - 100, 24, 24)
    self.playerBallView.frame=self.ballRect
    self.strokeCounter=0
    self.StrokeCount.text="杆数: \(strokeCounter) "
}else{
    .....
}
```

此时运行程序，会看到如图 9.14 所示的效果。

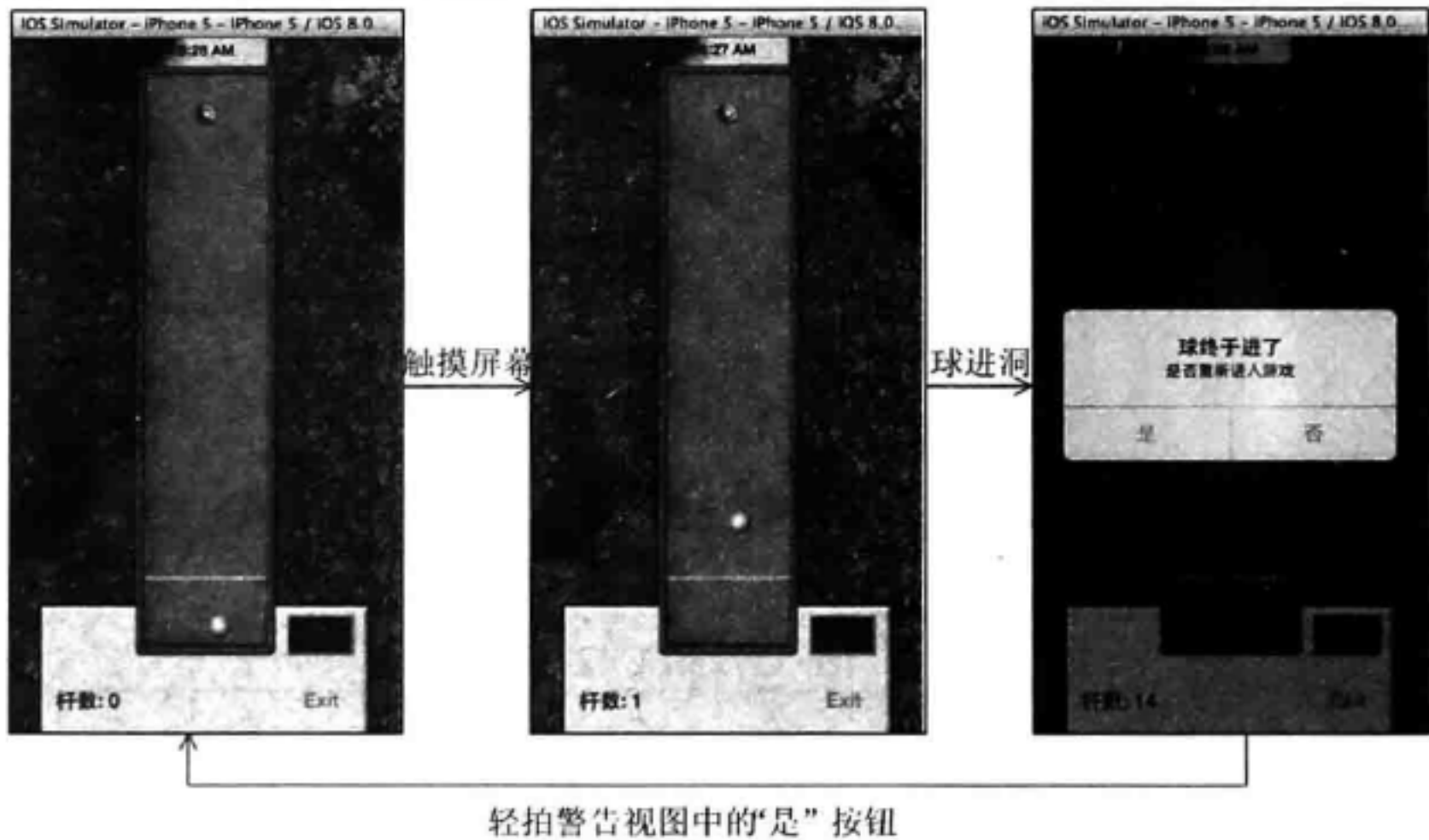


图 9.14 运行效果

9.7 游戏界面模块

本节将讲解关于游戏介绍界面的设计。在视图对象库中拖动 View Controller 视图控制器对象到画布中，然后对此视图控制器的界面进行设计，效果如图 9.15 所示。

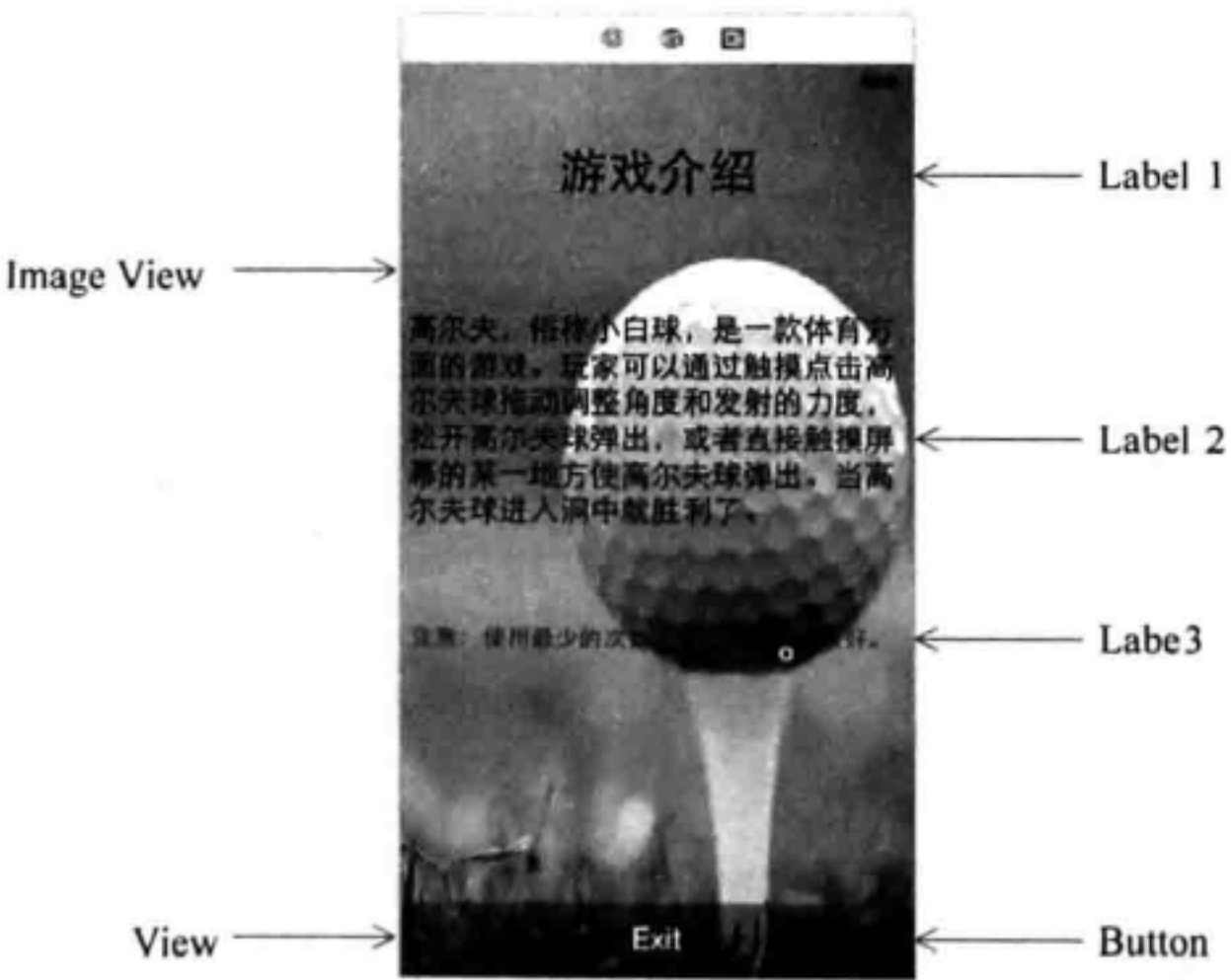


图 9.15 运行效果

需要添加的视图对象，以及对它们的设置，如表 9-4 所示。

表 9-4 设置界面

视 图	设 置
Image View	Image: Background.png 位置和大小: (0, 0, 320, 568)
Label1	Text: 游戏介绍 Font: System Bold 31.0 Alignment: 居中 位置和大小: (77, 40, 166, 53)
Label2	Text: 高尔夫，俗称小白球，是一款体育方面的游戏。玩家可以通过触摸点击高尔夫球拖动调整角度和发射的力度，松开高尔夫球弹出，或者直接触摸屏幕的某一地方使高尔夫球弹出。当高尔夫球进入洞中就胜利了。 Font: System 19.0 Lines: 6 位置和大小: (6, 148, 309, 148)
Label3	Text: 注意：使用最少的次数将球打入洞内为最好。 Color: 红色 Font: System 15.0 位置和大小: (6, 337, 309, 42)
Button	Title: Exit Font: System 19.0 Text Color: 白色 位置和大小: (81, 8, 159, 30)
View	Alpha: 0.6 Background: 深灰色 位置和大小: (0, 522, 320, 46)


9.8 场 景 切 换

在此游戏中需要将所有的场景进行切换，具体的切换关系如表 9-5 所示。

表 9-5 场景切换

对 象	切 换 到
Play Game 按钮	游戏界面
Game Description 按钮	游戏介绍界面
游戏界面中的 Exit 按钮	主菜单界面
游戏介绍界面中的 Exit 按钮	

最后画布中的效果如图 9.16 所示。

 **注意：**在实现场景切换前，首先需要单击实现主菜单的视图控制器，在此视图控制器中，选择界面上方的 Dock 中的 View Controller 图标，在工具窗口中的 Show the Attributes inspector 选项，即属性查看器选中，找到 Is Initial View Controller 复选框，将其选中，使此视图控制器成为初始视图控制器。

此时运行程序，可以看到以下的效果。当玩家在主菜单中轻拍 Play Game 按钮，可以

进入游戏界面。当玩家轻拍 Exit 按钮可以返回主菜单，效果如图 9.17 所示。

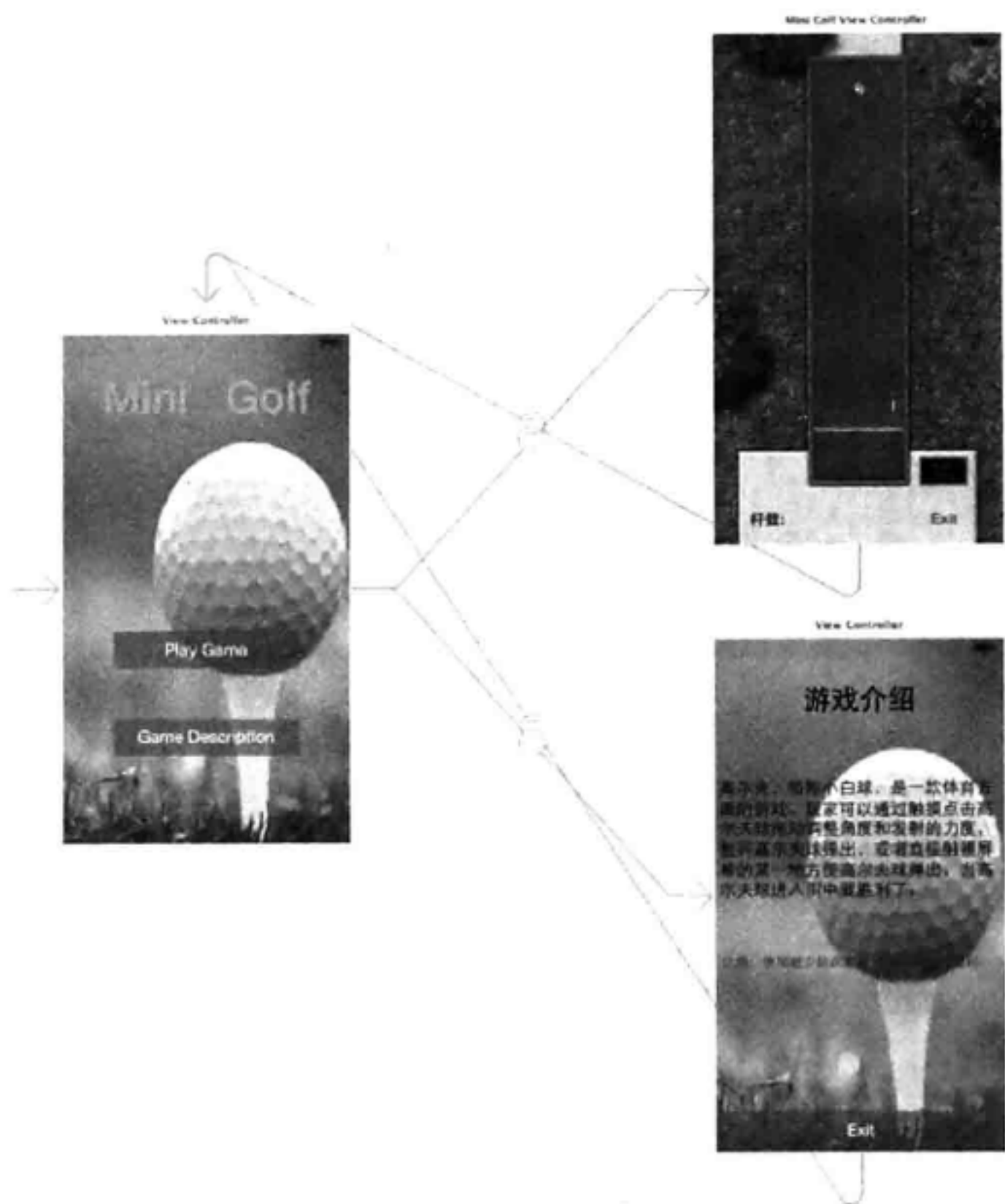


图 9.16 画布效果



图 9.17 运行效果

当玩家在主菜单中轻拍 Game Description 按钮，可以进入游戏介绍界面。当玩家轻拍 Exit 按钮，可以返回到主菜单，效果如图 9.18 所示。



图 9.18 运行效果

注意：场景的切换不仅可以使⤵用按钮来实现，也可以通过使用手势来实现。iOS 中常用的手势有 6 种，如轻拍、捏、旋转、滑动等。iOS 将常用的手势封装到一个 UITapGestureRecognizer 类中。这个类被称为手势识别器，一个手势就对应了一个手势识别器，如表 9-6 所示。

表 9-6 手势与手势识别器

手 势	手势识别器
轻拍	UITapGestureRecognizer
捏	UIPinchGestureRecognizer
滑动	UISwipeGestureRecognizer
旋转	UIRotationGestureRecognizer
移动	UIPanGestureRecognizer
长按	UILongPressGestureRecognizer

以下是通过滑动手势实现场景切换的步骤。
打开 ViewController.swift 文件，编写代码，通过滑动手势实现场景的切换功能。代码如下：

```
import UIKit
class ViewController: UIViewController {
    var vc:MiniGolfViewController=MiniGolfViewController()
    override func viewDidLoad() {
        super.viewDidLoad()
        //实例化并设置 swipL 对象
        varswipL:UISwipeGestureRecognizer=UISwipeGestureRecognizer(target:
            self,action: Selector("left")) //实例化对象
        swipL.direction=UISwipeGestureRecognizerDirection.Left //设置方向
        self.view.addGestureRecognizer(swipL) //添加手势识别器
    }
    ..... //这里省略了 didReceiveMemoryWarning() 方法
    //实现界面的切换
    func left(){
        var transiton=CATransition()
```



```
transiton.duration=5.0
transiton.type=kCATransitionPush
transiton.subtype=kCATransitionFromRight
vc=self.storyboard!.instantiateViewControllerWithIdentifier("pvc")
as MiniGolfViewController
self.view.addSubview(vc.view)
self.view.layer.addAnimation(transiton, forKey: nil)
}
```

此时运行程序，会看到如图 9.19 所示的效果。

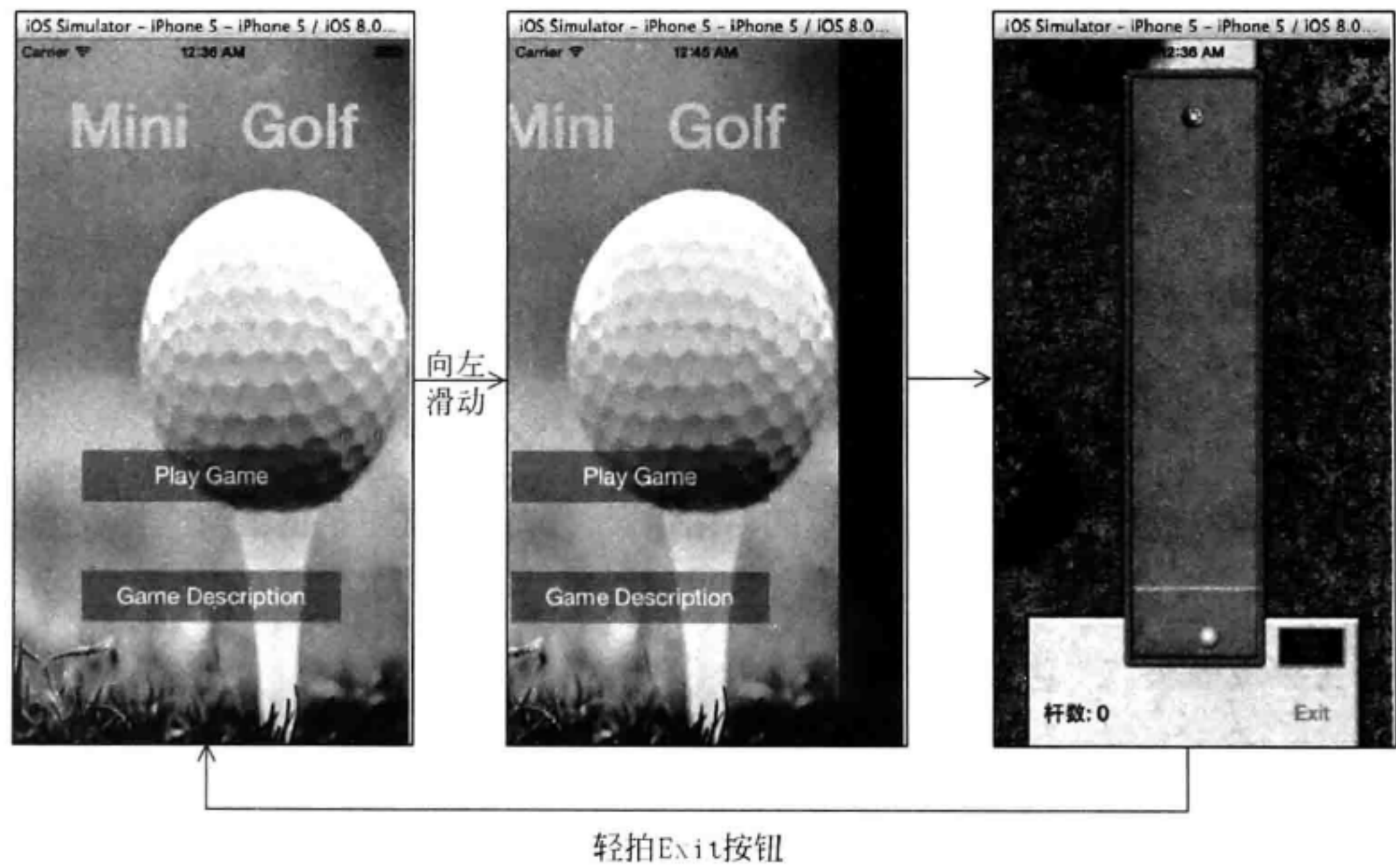


图 9.19 运行效果

第10章 银河大战——Sprite Kit 游戏引擎和传感器应用

银河大战和太空侵略者的游戏类似，也是射击游戏的一种。但是银河大战这款游戏是通过使用 Sprite Kit 游戏引擎来制作的，同时还使用到了传感器。本章将通过这个游戏来讲解 Sprite Kit 游戏引擎以及传感器的技术。

10.1 游戏介绍

银河大战游戏是一款既经典又好玩的射击游戏。如果玩家在宇宙银河或者太空星际中遇到敌人都必须将它们全部消灭。这样一款游戏，包括以下几个模块。

1. 主菜单模块

在主菜单的界面上有两个菜单项，如图 10.1 所示。当玩家轻拍 Play Game 菜单项，进入射击游戏界面；轻拍 Game Description 菜单项，进入游戏介绍界面。

2. 游戏模块

游戏模块提供了游戏的界面，如图 10.2 所示。玩家可以在此界面中进行射击游戏。



图 10.1 主菜单模块



图 10.2 游戏模块

3. 游戏介绍模块

游戏介绍模块提供了对游戏的介绍以及玩法，如图 10.3 所示。



图 10.3 游戏介绍模块

10.2 创建 Game 类型项目

在开发银河大战游戏之前，需要做一些准备工作，这些准备工作如下所述。

10.2.1 Game 模板类型简介

在 Xcode 6 之前，是没有 Game 模板的，如图 10.4 所示。

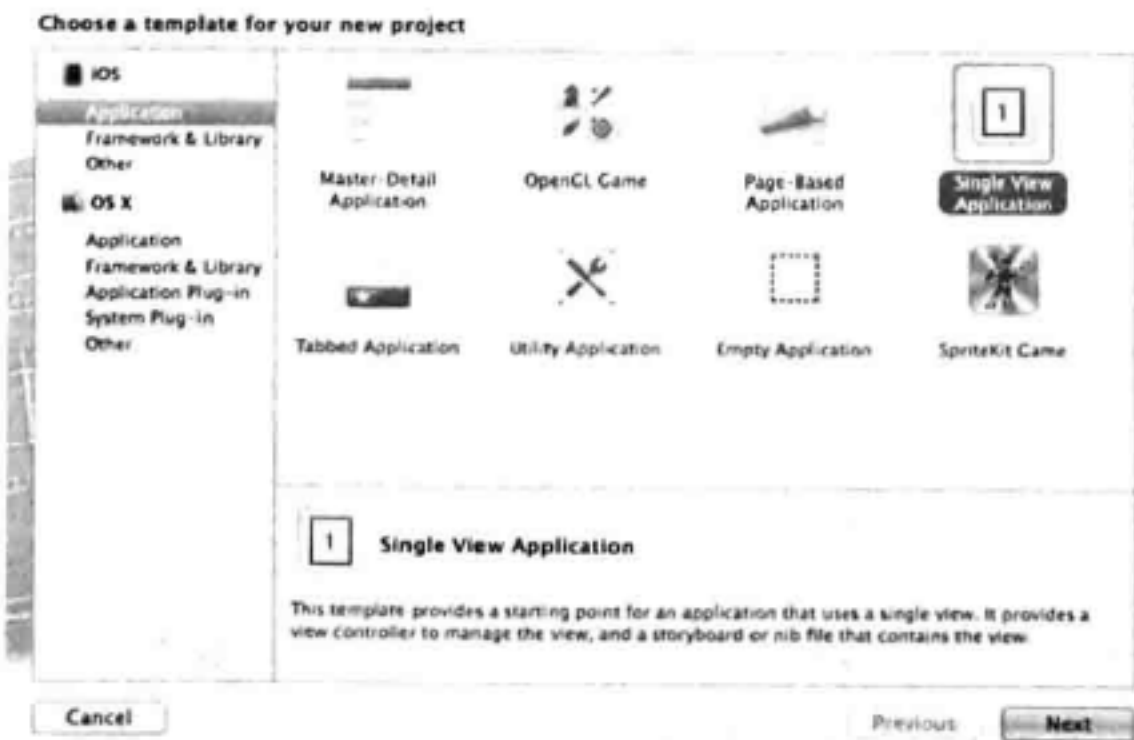


图 10.4 模板类型

此图是 Xcode 5 的模板类型。而 Game 模板就是将 SpriteKit 技术、OpenGL ES 技术，以及新的 Metal 技术放在一起的一个模板，开发者可以在此模板中选择自己所需的技术。在本章中，我们将使用该模版来构建银河大战的游戏。

10.2.2 创建项目

和其他的 iOS 应用程序一样，在开发这款游戏之前，首先需要创建一个项目。本项目将基于 Game 模板构建。此项目创建的具体步骤如下所述。

- (1) 单击 Dock 中的 Xcode，弹出 Welcome to Xcode 对话框。
- (2) 选择其中的 Create a new Xcode project，弹出 Choose a template for your new project:对话框，如图 10.5 所示。

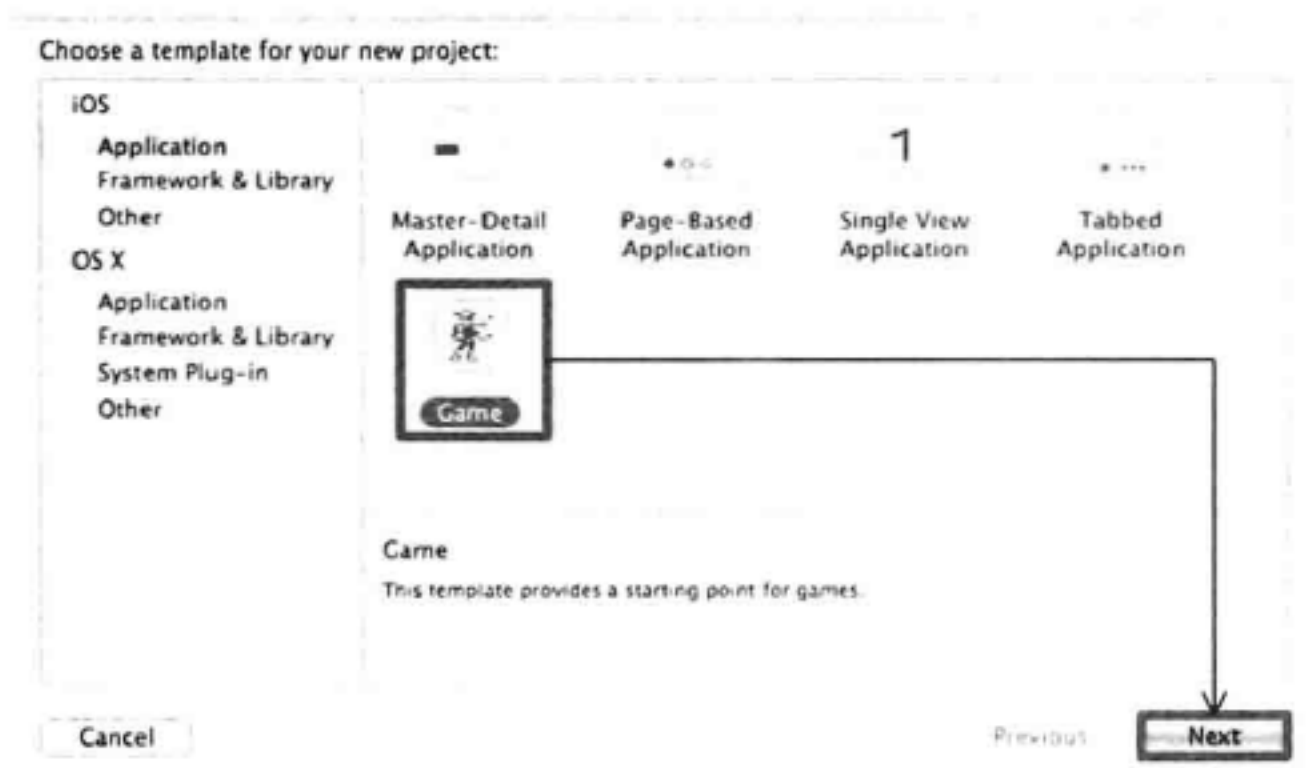


图 10.5 Choose a template for your new project:对话框

- (3) 选择 iOS|Application 中的 Game 模板，然后单击 Next 按钮，弹出 Choose options for your new project:对话框，如图 10.6 所示。

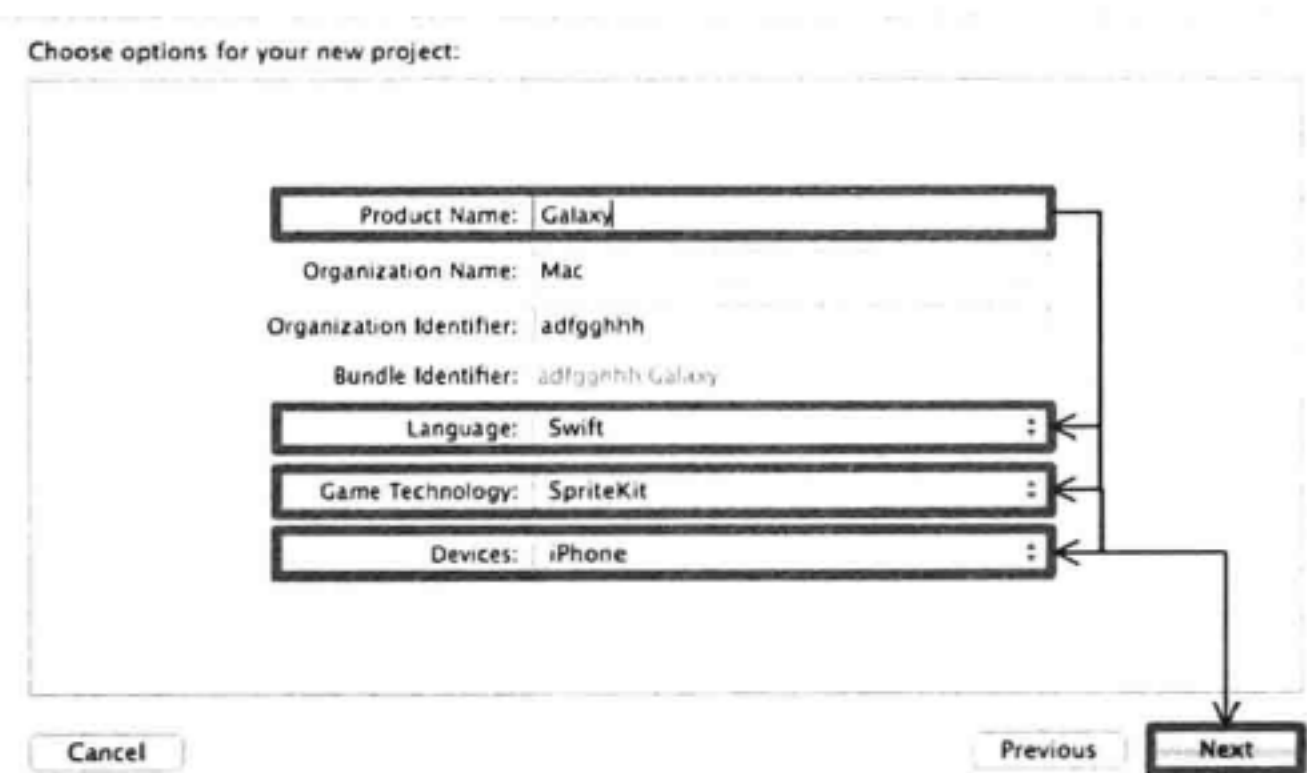


图 10.6 Choose options for your new project:对话框

(4) 输入项目名称，将语言改为 Swift，将游戏的技术改为 SpriteKit，将设备改为 iPhone。然后单击 Next 按钮，弹出选择项目保存位置的对话框。单击 Create 按钮后，一个名为 Galaxy 的项目就创建好了。我们就可以在此项目中实现银河大战游戏了。

注意：此时创建好的 Galaxy 项目和之前使用 Single View Controller 模板创建的项目是有区别的。在 Galaxy 项目中多了一个.swift 的文件，并且有事先编写好的代码。运行此代码，可以看到如图 10.7 所示的效果。

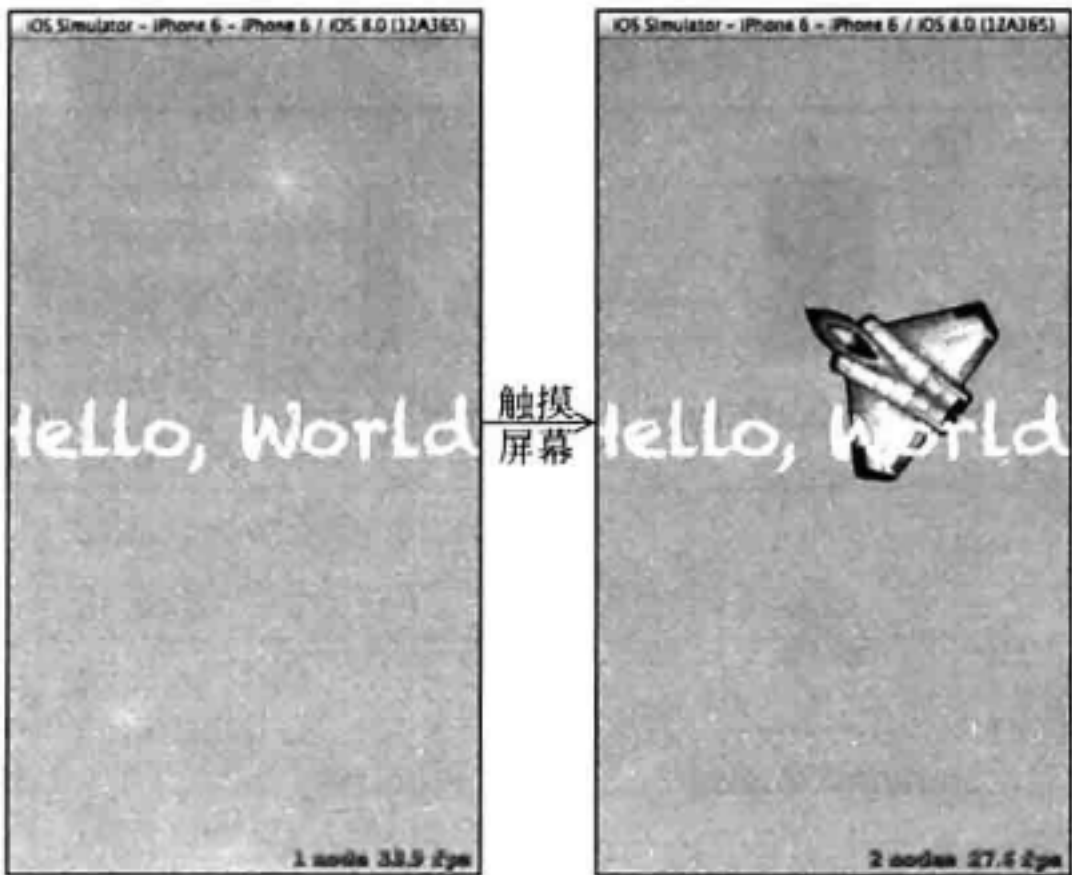


图 10.7 运行效果

在触摸屏幕后会出现旋转的飞船。最下方的内容为游戏的帧数，开发者可以通过对 `showsFPS` 属性的设置对游戏的帧数进行显示和隐藏：如果需要将图 10.7 中游戏的帧数隐藏，开发者可以打开 `GameViewController.swift` 文件，在 `viewDidLoad()` 方法中找到以下的代码：

```
skView.showsFPS = true
```

将此代码改为以下的代码：

```
skView.showsFPS = false
```

此时运行程序，会看到如图 10.8 所示的效果。

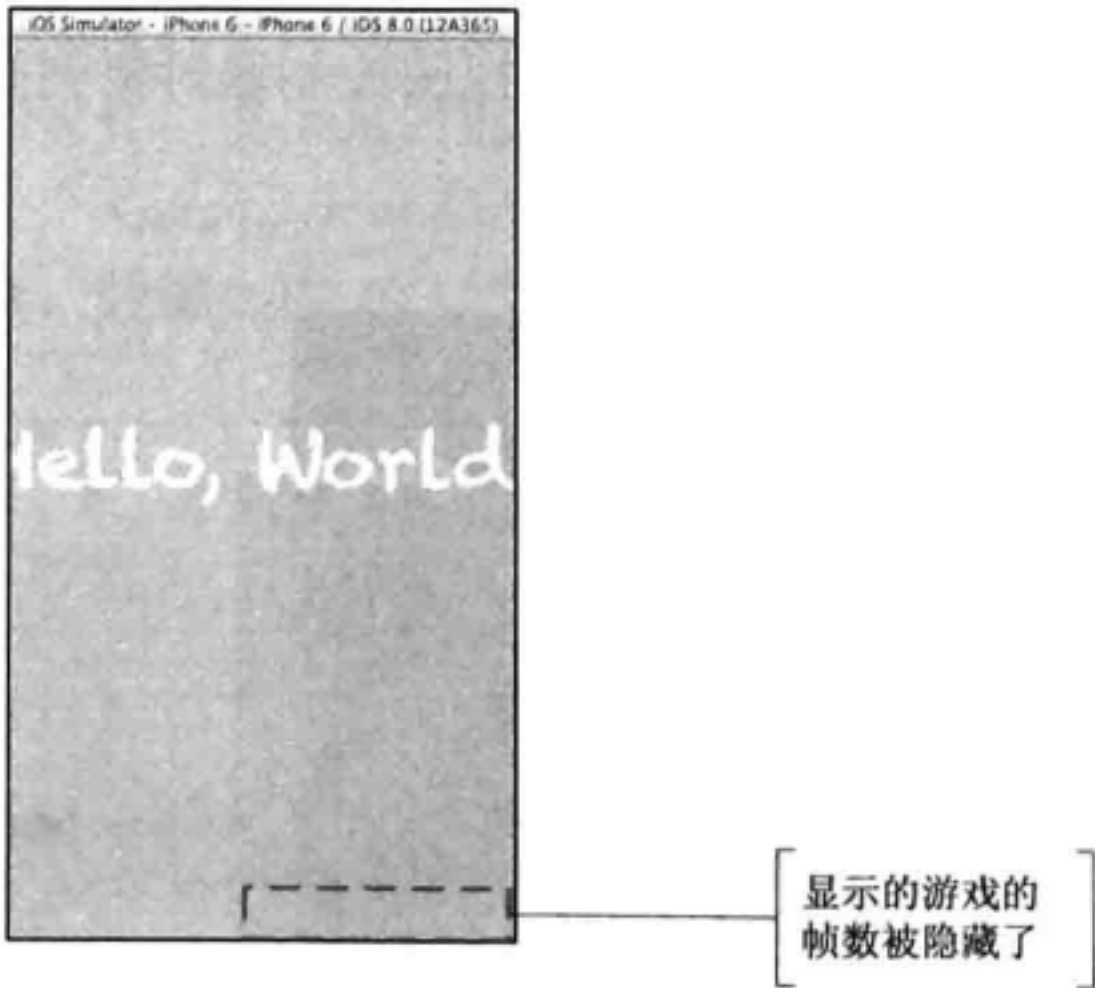


图 10.8 运行效果

10.2.3 添加图像和音频文件

创建好项目之后，还需要添加一些文件。这里将添加的文件分为了两种：一种是图像文件；另一种是音频文件。添加图像 `background1.png`、`background2.png`、`bullet.png`、`enemy01.png`、`enemy02.png`、`enemy03.png`、`logo.png` 和 `ship.png` 到创建项目的 Supporting Files

文件夹中，如图 10.9 所示。

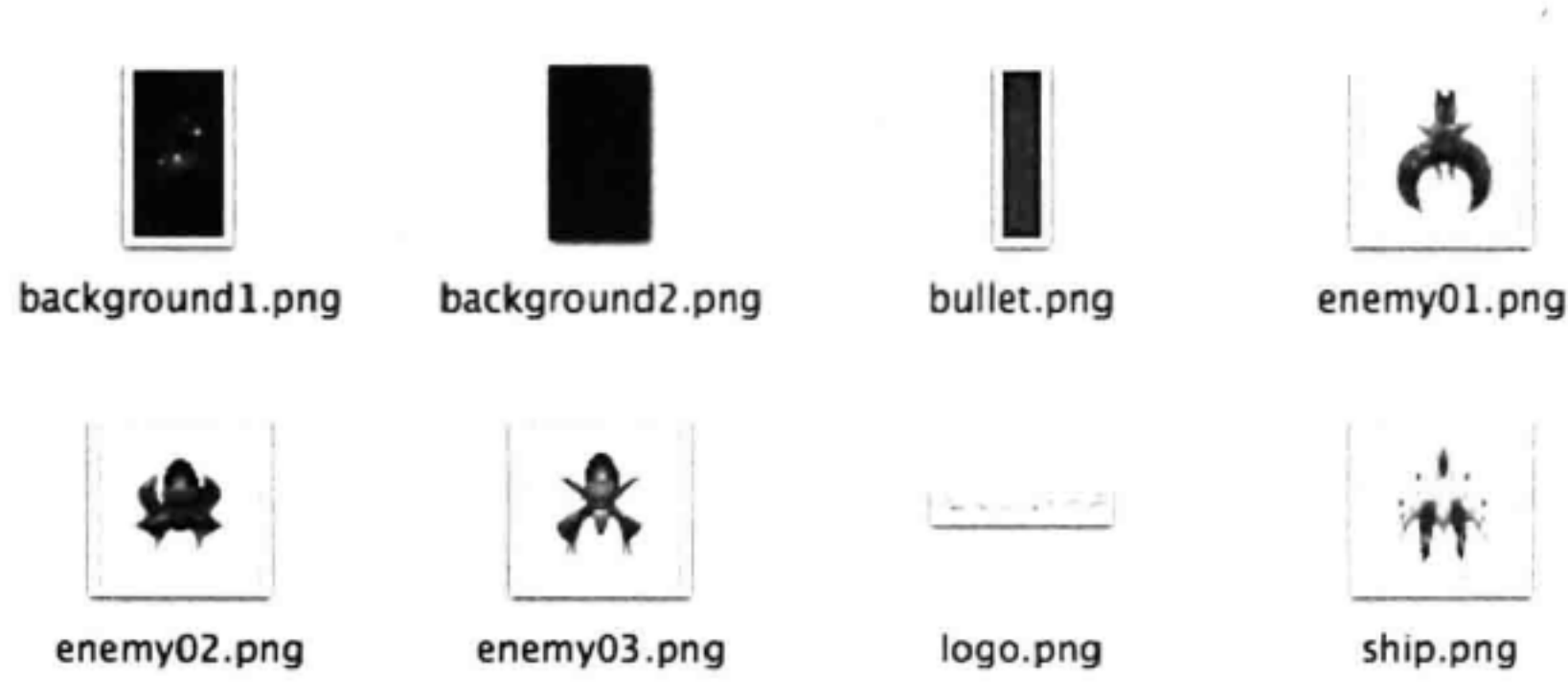


图 10.9 添加的图像

添加音频文件 `impact.wav` 和 `startup.wav` 到创建项目的 Supporting Files 文件夹中。

10.3 主菜单模块

在本章的游戏中，存在一个主菜单，在主菜单的界面中有两个菜单项。这两个菜单项都是使用按钮实现的。本节将讲解此游戏的主菜单的设计。双击将 `Main.storyboard` 文件打开，对主菜单的界面进行设计，具体的操作步骤如下所述。

- (1) 选择 `Show the File inspector` 选项，将 `Interface Builder Document` 中的 `Opens in` 改为 `Xcode 5.1`。
- (2) 在视图对象库中拖动 `View Controller` 视图控制器对象到画布中。对此视图控制器的界面进行设计，效果如图 10.10 所示。

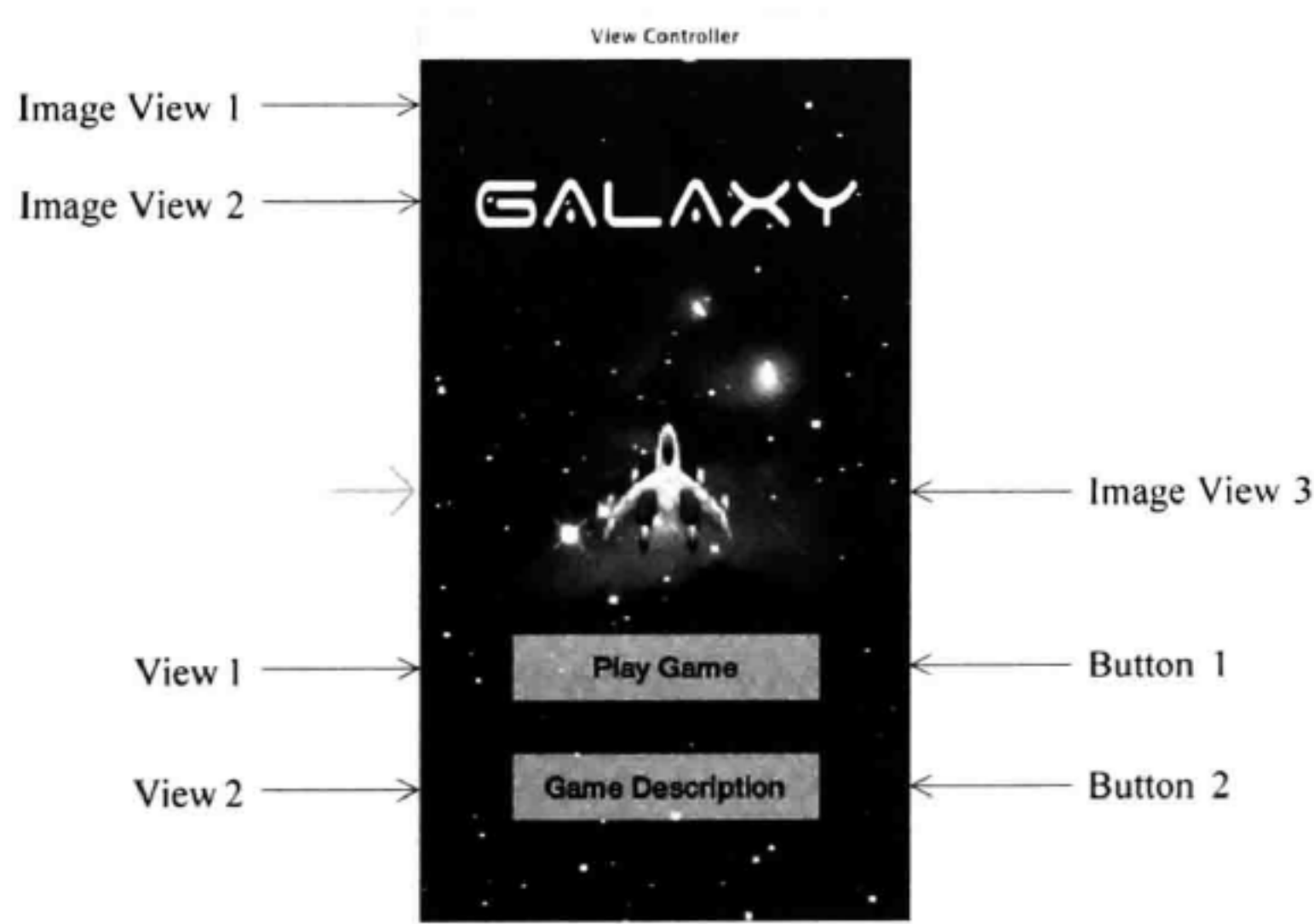


图 10.10 界面效果

需要添加的视图对象，以及对它们的设置，如表 10-1 所示。

表 10-1 设置界面

视 图	设 置
Image View1	Image: background1.png 位置和大小: (0, 0, 320, 568)
Image View2	Image: logo.png 位置和大小: (31, 65, 259, 59)
Image View3	Image: empty.png 位置和大小: (88, 227, 145, 113)
View1	Alpha: 0.6 位置和大小: (60, 378, 201, 44)
Button1	Title: Play Game Font: System Bold 19.0 Text Color: 黑色 位置和大小: (46, 8, 109, 30)
View2	Alpha: 0.6 位置和大小: (60, 458, 201, 44)
Button2	Title: Game Description Font: System Bold 19.0 Text Color: 黑色 位置和大小: (18, 8, 165, 30)

10.4 射击游戏模块

在一个游戏中，游戏模块或者实现游戏功能的模块都是游戏的核心，它提供了游戏界面，可以让玩家进行一些操作。以下是对射击游戏模块的界面进行的射击。

在画布中原本就有一个 Game View Controller 视图控制器，我们需要在此视图控制器上实现对游戏界面的设计，效果如图 10.11 所示。

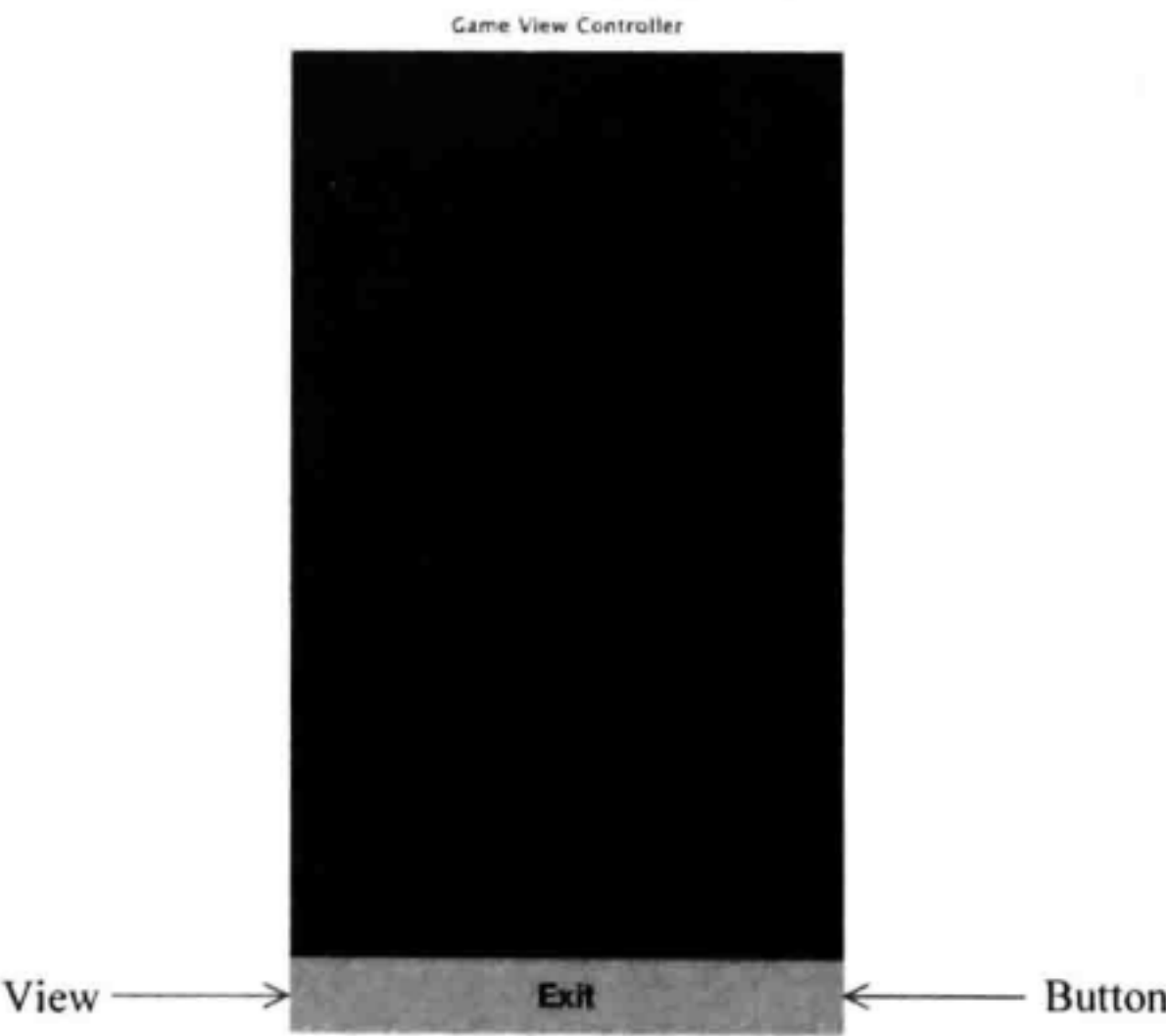


图 10.11 界面效果

需要添加的视图对象，以及对它们的设置，如表 10-2 所示。

表 10-2 设置界面

视 图	设 置
View	Alpha: 0.6 位置和大小: (0, 525, 320, 43)
Button	Title: Exit Font: System Bold 19.0 Text Color: 黑色 位置和大小: (137, 7, 46, 30)

10.5 为射击游戏界面添加元素

在图 10.11 中可以看到，在射击游戏的界面中，除了有一个 Exit 按钮外，没有任何其他的内容。本节将讲解为游戏的界面添加元素，这些元素包括飞船和敌人等。

10.5.1 准备工作

为了方便游戏的开发，我们需要在 GameScene.swift 文件和 GameViewController.swift 文件中编写银河大战的游戏。在编写代码之前，首先需要将这两个文件中的一些内容进行删除。

1. GameScene.swift

打开 GameScene.swift 文件，将文件中的一些方法删除，删除后的文件的剩余的内容如图 10.12 所示。

```
import SpriteKit

class GameScene: SKScene {
    override func didMoveToView(view: SKView) {

    }


    override func touchesBegan(touches: NSSet, withEvent event: UIEvent) {

    }

    override func update(currentTime: CFTimeInterval) {

    }
}
```

图 10.12 GameScene.swift 文件中的内容

 **注意：**GameScene 类是继承 SKScene 类的。SKScene 类被称为场景节点，它将作为根节点存在，而其他的内容节点都是其子节点。场景节点将负责运行一个动画循环，从而来处理其他子节点的动作，模拟物理世界，并将节点数中的内容渲染到屏幕上。Sprite Kit 是由场景（Scene）来组成的。

2. GameViewController.swift

打开 GameViewController.swift 文件，将文件中的一些方法删除，删除后的文件的剩余的内容如图 10.13 所示。

```
import UIKit
import SpriteKit

class GameViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
    }

}
```

图 10.13 GameViewController.swift 文件中的内容

10.5.2 什么是 Sprite Kit

Sprite Kit 技术是 iOS 7 内置的一个新的框架，该框架主要用来开发 2D 游戏。目前已经支持的内容包括：精灵、酷炫特效（例如视频、滤镜和遮罩），并且还集成了物理引擎库等许多东西。在 Sprite Kit 框架下封装了许多类，在银河游戏中将使用到 SKSpriteNode、SKAction 和 SKLabelNode 这些类。

10.5.3 使用 SKSpriteNode 添加背景

在对射击游戏界面进行设计时，我们没有进行背景的添加。那是因为在 Sprite Kit 技术中不允许使用 UIKit 中的 Image View 图像视图对背景进行添加，而需要使用 SKSpriteNode 类。此类的使用需要实现以下几个步骤。

1. 实例化对象

SKSpriteNode 是一个类，所以在使用它的时候需要进行实例化。SKSpriteNode 实例化的方法有很多，最常使用的是 SKSpriteNode(imageNamed: String)方法，其语法形式如下：

```
var SKSpriteNode 对象名:SKSpriteNode= SKSpriteNode(imageNamed: String)
```

其中，imageName 用来指定绘制精灵纹理，即图像。

2. 设置位置

创建好 SKSpriteNode 对象后，需要使用 position 属性对此对象的位置进行设置，其语法形式如下：

```
对象名.position=位置
```

3. 添加到场景场景中

最后需要使用 addChild(node:SKNode)方法将 SKSpriteNode 对象添加到场景节点中，其语法形式如下：


```
addChild(node:SKNode)
```

下面就添加背景的具体步骤。

(1) 打开 GameScene.swift, 在 didMoveToView(view: SKView)方法中编写代码, 实现对背景的添加, 代码如下:

```
override func didMoveToView(view: SKView) {
    var background:SKSpriteNode=SKSpriteNode(imageNamed: "background2.png")
    //创建对象
    //设置位置
    background.position=CGPointMake(CGRectGetMidX(self.frame), CGRectGetMidY(self.frame))
    self.addChild(background) //添加到场景节点中
}
```

(2) 打开 GameViewController.swift 文件, 将场景显示在界面中, 代码如下:

```
override func viewDidLoad() {
    super.viewDidLoad()
    let skView = self.view as SKView //将 view 转换为 SKView
    skView.showsFPS = true //游戏的帧数进行显示
    skView.showsNodeCount = true //显示节点数量
    skView.ignoresSiblingOrder = true
    let scene = GameScene(size: skView.bounds.size) //实例化场景对象
    scene.scaleMode = .AspectFill
    skView.presentScene(scene) //显示场景
}
```

此时运行程序, 会看到如图 10.14 所示的效果。

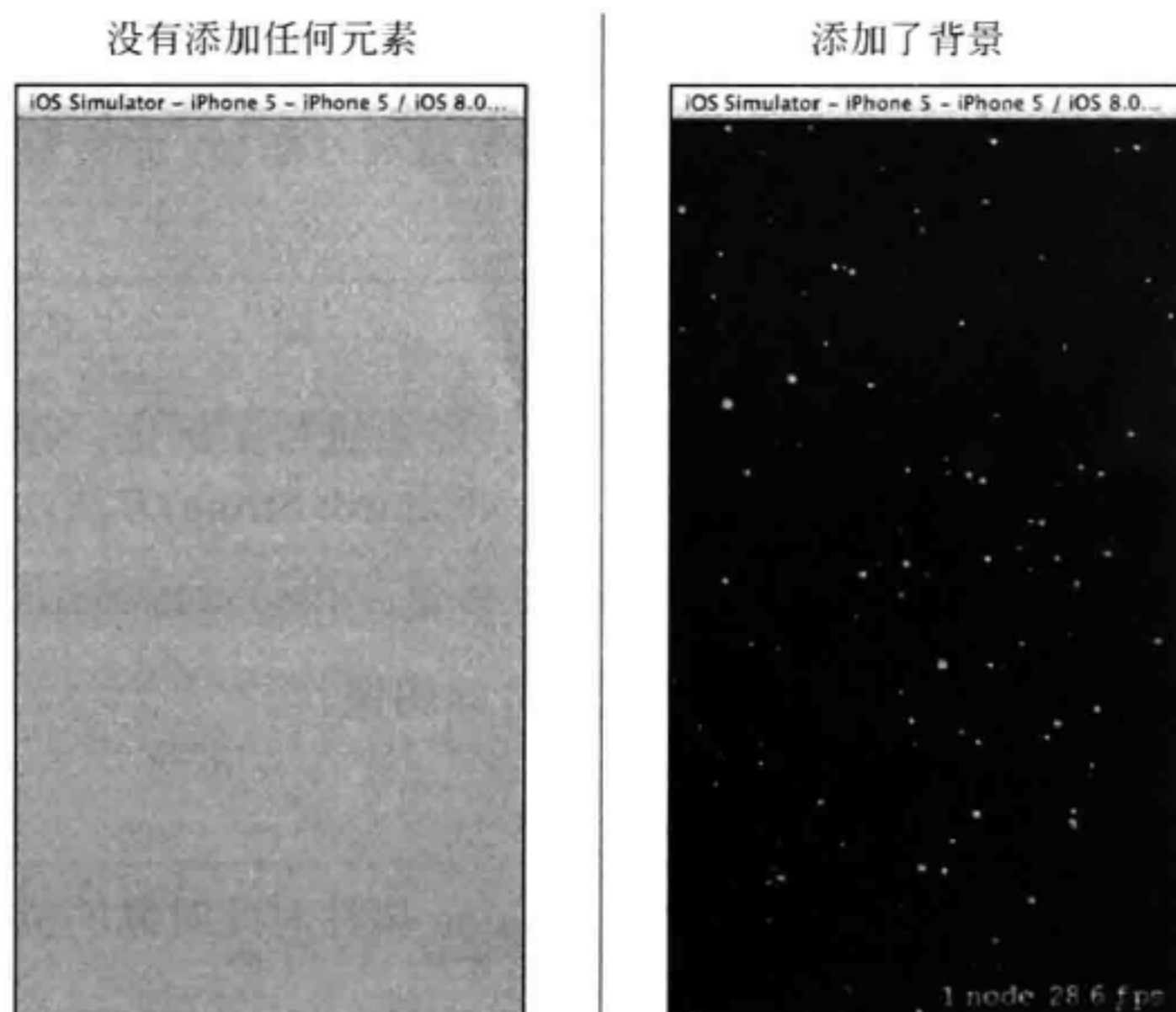


图 10.14 运行效果

注意: 在 Sprite Kit 中的所有视觉元素均使用 SKNode 的子类来绘制。SKNode 大致包括的类如表 10-3 所示。

表 10-3 SKNode大致包括的类

类	功 能
SKSpriteNode	用于绘制精灵纹理
SKVideoNod	用于播放视频
SKLabelNode	用于渲染文本
SKShapeNode	用于渲染基于 Core Graphics 路径的形状
SKEmitterNode	用于创建和渲染粒子系统
SKCropNode	用于使用遮罩来裁剪子节点
SKEffectNode	用于在子节点上使用 Core Image 滤镜

注意: SKView 类是专门用来呈现 Sprite Kit 的 View, 在此类中可以渲染和管理一个 SKScene, 每个 Scene 中可以加载多个精灵, 并管理它们的行为。

10.5.4 使用 SKSpriteNode 添加飞船

在银河大战中也存在着让玩家进行操控的飞船, 此飞船代表的是正义的一方。它的添加同样还是需要使用到 SKSpriteNode 类, 以下就是添加飞船的具体的步骤。

(1) 创建一个 SKSpriteNode 的实例对象, 在此对象中用来显示飞船, 代码如下:

```
var playerShip:SKSpriteNode=SKSpriteNode(imageNamed: "ship.png")
```

(2) 在 didMoveToView(view: SKView)方法中实现 SKSpriteNode 对象的位置设置以及添加, 代码如下:

```
self.addChild(background)
self.playerShip.position = CGPointMake(CGRectGet MidX(self. frame),CGRect
GetMinY(self.frame) + self.playerShip.frame.size.height)
self.addChild(self.playerShip)
```

此时运行程序, 会看到如图 10.15 所示的效果。

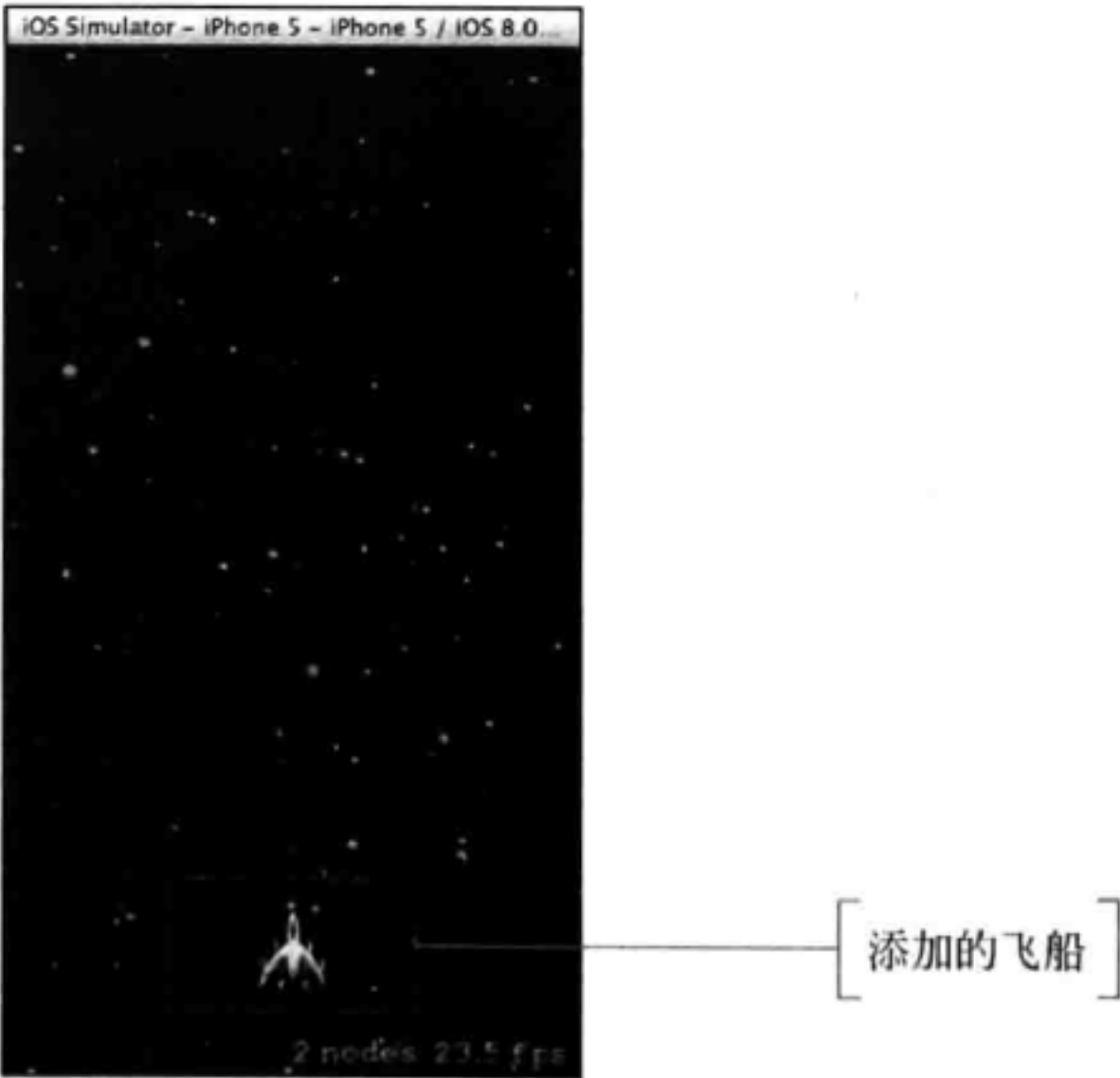


图 10.15 运行效果

10.5.5 使用 SKSpriteNode 添加敌人

在银河大战游戏中存在正义的一方，自然就会存在非正义的一方，即敌人。敌人的添加方式和添加背景与飞船的方式类似，只不过由于敌人的数量很多，需要使用到 for 循环。以下就是添加敌人的具体步骤。

(1) 声明并定义两个变量，代码如下：

```
var enemyCount:Int=5 //敌人的数目
var enemies:NSMutableArray=NSMutableArray(capacity: 5) //保存敌人的数组
```

(2) 在 didMoveToView(view: SKView)方法中实现敌人的添加，代码如下：

```
self.addChild(self.playerShip)
//添加敌人
for(var i=0;i<enemyCount;++i){
    var enemyName:String="enemy0\((random() % 3)+1).png"
    var enemy=SKSpriteNode(imageNamed: enemyName) //实例化对象
    enemy.position = CGPointMake(25 + (CGFloat(i)*enemy.size.width), 400) //设置对象的位置
    enemy.hidden=false
    self.enemies.addObject(enemy) //为数组添加对象
    self.addChild(enemy) //为场景节点添加节点
}
```

此时运行程序，会看到如图 10.16 所示的效果。

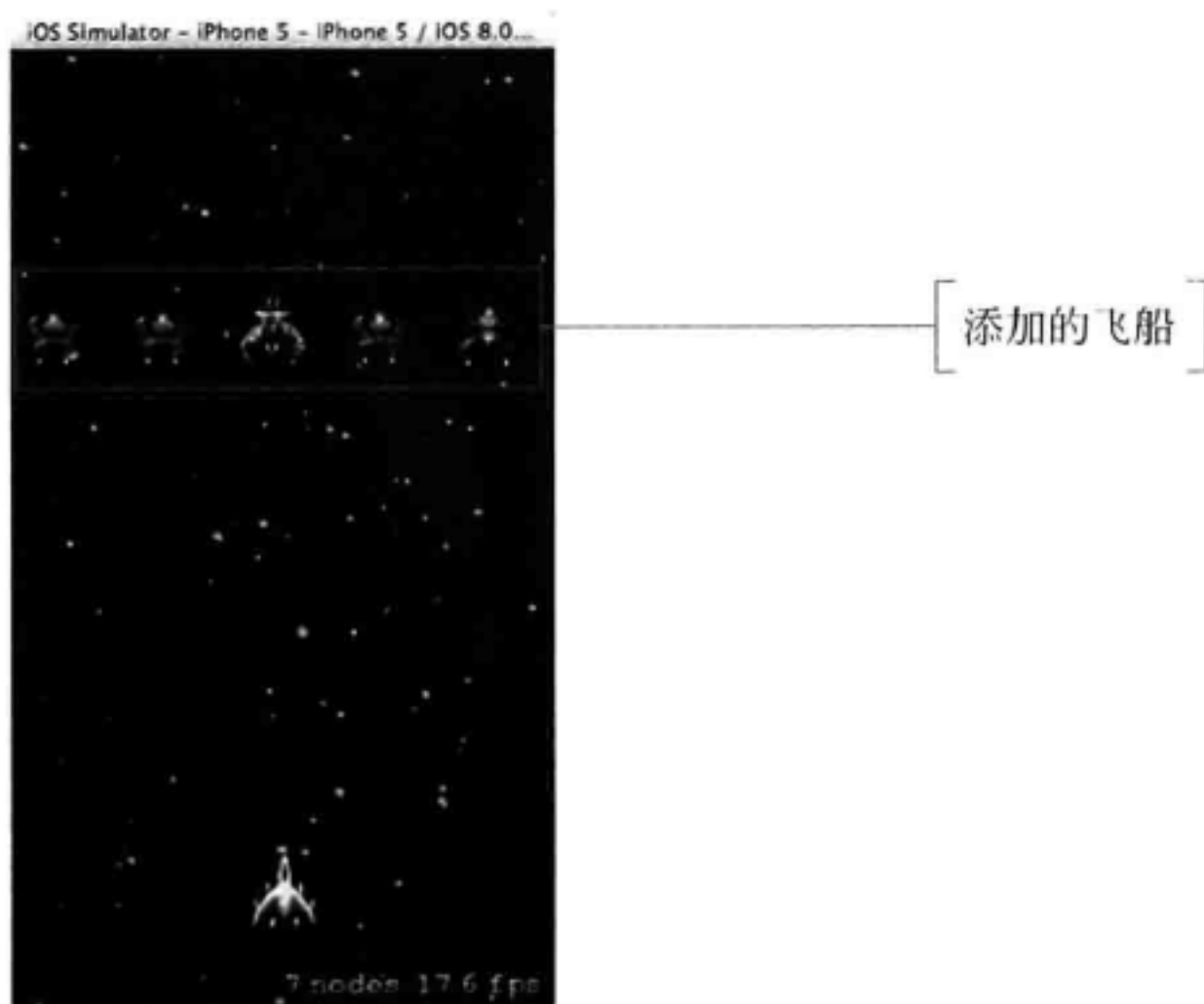


图 10.16 运行效果

10.6 发射子弹

和太空侵略者的游戏一样，飞船需要发射子弹，通过子弹才可以将敌人消灭。本节将讲解子弹的发射功能。

10.6.1 添加子弹

在发射子弹之前，首先需要实现对子弹的添加，添加子弹的方式需要使用到 SKSpriteNode 类。具体的实现步骤如下所述。

(1) 声明两个变量，代码如下：

```
var currentBullet: Int = 0 //当前的子弹
var playerBullets: NSMutableArray = NSMutableArray(capacity: 5) //用来保存子弹的数组
```

(2) 在 didMoveToView(view: SKView)方法中实现子弹的添加，类似于敌人的添加，代码如下：

```
for (var i = 0; i < enemyCount; ++i) {
    var bullet = SKSpriteNode(imageNamed: "bullet.png") //实例化对象
    bullet.position = self.playerShip.position //设置位置
    bullet.hidden = true
    self.playerBullets.addObject(bullet)
    self.addChild(bullet) //添加节点
}
```

10.6.2 通过触摸发射子弹

对子弹的发射我们使用的是通过触摸屏幕的方式，所以需要重写 touchesBegan(touches: NSSet, withEvent event: UIEvent)方法，在此方法中实现子弹的发射效果，代码如下：

```
override func touchesBegan(touches: NSSet, withEvent event: UIEvent) {
    var touch: UITouch = UITouch()
    //遍历
    for touch in touches {
        var playerBullet: SKSpriteNode = self.playerBullets.objectAtIndex(
            currentBullet) as SKSpriteNode
        currentBullet++
        //判断当前的子弹是否大于 playerBullets 的个数
        if (currentBullet >= self.playerBullets.count) {
            currentBullet = 0
        }
        playerBullet.position = self.playerShip.position //发射子弹的位置
        playerBullet.hidden = false
        var fireBulletAction: SKAction = SKAction.moveTo(CGPointMake(self.
            playerShip.position.x, self.frame.size.height), duration: 1)
        //实现子弹的移动
        //结束子弹的发射
        var endBulletAction: SKAction = SKAction.runBlock({
            playerBullet.removeAllActions() //移除所有的动作
            playerBullet.hidden = true //隐藏发射的子弹
        })
        //将发射子弹和结束子弹的发射这两个动作组合
        var fireBulletAndDestroy: SKAction = SKAction.sequence([fireBulletAction,
            endBulletAction])
        playerBullet.runAction(fireBulletAndDestroy, withKey: "Fire")
        //运行动作
    }
}
```

```
}  
}
```

⚠注意：对于子弹发射的动画，我们使用了 SKAction 类。它提供了一切的动作。如平移、旋转，以及组合动作等。

此时运行程序，可以看到如图 10.17 所示的效果。

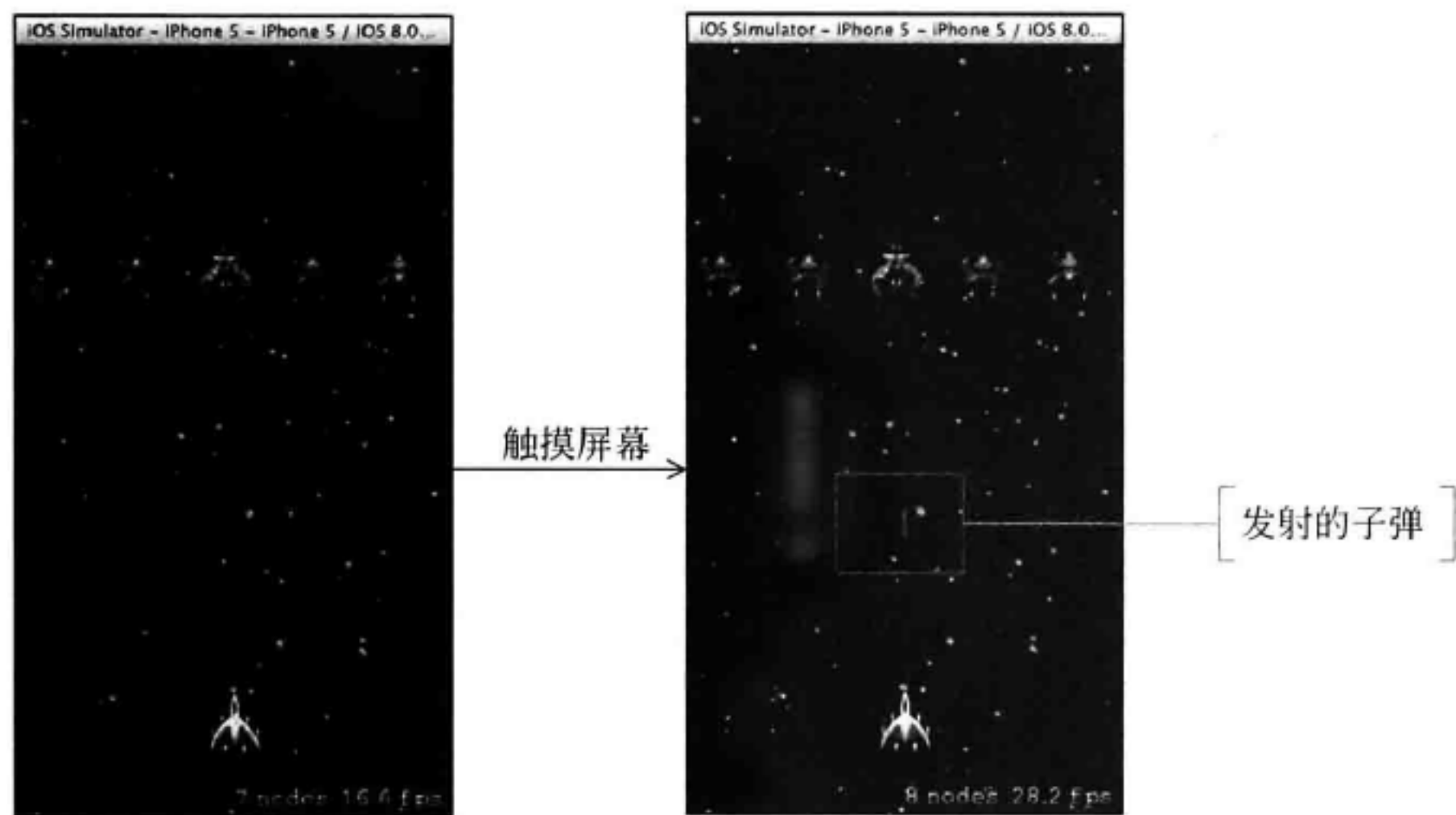


图 10.17 运行效果

10.7 使用传感器操控飞船

和太空侵略者的游戏类似，此游戏的飞船也不是一直静止的，它也需要进行移动。本节将讲解通过传感器让飞船移动，即当设备向左倾斜时，飞船向左移动，当设备向右倾斜时，飞船向右移动。

10.7.1 传感器介绍

传感器是一种检测装置，能感受到被测量的信息，并能将感受到的信息以某种形式输出。在 iPhone 设置中用到的传感器有影像传感器、亮度传感器、磁阻传感器、距离传感器、声波传感器、加速度传感器、角加速度传感器。在需要重力感应的应用程序中，加速度传感器和角度加速度传感器是十分常见的。这两种感应器通过公开的 API 中的 CMMotionManager 类实现相应的功能。在我们的游戏中，使用到了重力感应，即向左倾斜和向右倾斜。

10.7.2 判断传感器是否可用

在使用传感器之前，首先需要对传感器进行判断，判断此设备上的传感器是否可用。判断传感器需要使用 accelerometerAvailable 方法，以下就实现判断的步骤。

(1) 导入 CoreMotion 框架，代码如下：

```
import CoreMotion
```

(2) 实例化一个 CMMotionManager 对象，代码如下：

```
var mManager:CMMotionManager?=CMMotionManager()
```

(3) 在 didMoveToView(view: SKView)方法中编写判断传感器是否可用的代码，代码如下：

```
//判断传感器是否可用
if((self.mManager?.accelerometerAvailable) == true){
    //可用
    var alert:UIAlertView=UIAlertView()
    alert.title="Accelerometer Available "
    alert.addButtonWithTitle("Cancel")
    alert.show()
}else{
    //不可用
    var alert:UIAlertView=UIAlertView()
    alert.title="Sorry,No Accelerometer"
    alert.addButtonWithTitle("Cancel")
    alert.show()
}
```

此时运行程序，可以看到以下的运行效果。当设备中的传感器可用时，会弹出 Accelerometer Available 的警告视图，如图 10.18 所示。当设备中的传感器不可用时，会弹出 Sorry,No Accelerometer 的警告视图，如图 10.19 所示。

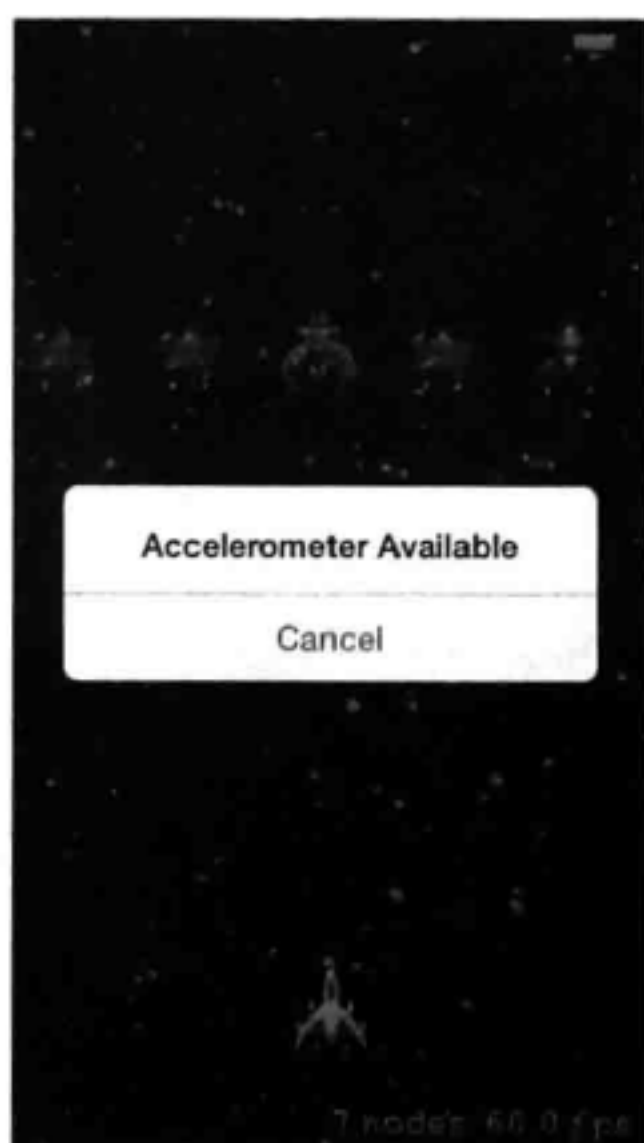


图 10.18 传感器可用



图 10.19 传感器不可用

10.7.3 实现移动

飞船的移动是通过传感器对设备的倾斜方向进行检测的。如果向左倾斜，飞船就向左

移动；如果向右倾斜，飞船就向右移动。它的实现思路如下所述。

首先，需要对设备的方向进行检测，需要使用 `CMMotionManager` 类。然后开启数据更新功能，将更新的内容放置在 `update(currentTime:CTimeInterval)` 方法中。最后，实现飞船的移动。

以下是实现飞船移动的具体步骤。

(1) 需要在判断传感器是否可用的代码中添加以下的代码，开启数据更新功能：

```
if((self.mManager?.accelerometerAvailable) == true){
    var alert:UIAlertView=UIAlertView()
    alert.title="Accelerometer Available "
    alert.addButtonWithTitle("Cancel")
    alert.show()
    self.mManager?.startAccelerometerUpdates()           //更新数据
}else{
    .....
}
```

(2) 为了让飞船更具真实感，需要为飞船添加一些物理现象。在 `didMoveToView(view:SKView)` 方法中添加以下代码（Sprite Kit 有一个基于 Box 2D 的强大的内置物理系统，Box 2D 可以模拟大量物理现象，如力、转化、旋转、碰撞和接触察觉。每一个 `SKNode` 和 `SKSpriteNode` 都有一个附属于它的 `SKPhysicsBody`。这个 `SKPhysicsBody` 表示物理模拟）：

```
//将物理体添加到场景中
self.physicsBody=SKPhysicsBody(edgeLoopFromRect:self.frame)
//创建一个矩形的物理体，大小与飞船一致
self.playerShip.physicsBody=SKPhysicsBody(rectangleOfSize:self.playerShip.frame.size)
self.playerShip.physicsBody?.dynamic=true           //能承受碰撞和其他外力作用
self.playerShip.physicsBody?.affectedByGravity=false //不受重力影响
self.playerShip.physicsBody?.mass=0.2               //给飞船任意质量，这样它的移动会显得更自然
```

(3) 重写 `update(currentTime:CTimeInterval)` 方法，在此方法中需要调用一个 `shipUpdates()` 的方法，代码如下：

```
//更新游戏中角色的信息
override func update(currentTime:CTimeInterval){
    self.shipUpdates()
}
```

(4) 添加一个 `shipUpdates()` 方法，在此方法中实现飞船的移动，代码如下：

```
func shipUpdates(){
    var data:CMAccelerometerData?=self.mManager?.accelerometerData
    //获取 accelerometer 数据
    var value:Double?=data?.acceleration.x
    //判断 value 是否为空
    if(value==nil){
        value=0
    }
    //判断设备倾斜的方向
    if(fabs(value!)>0.2)
    {
        var fvector=CGVectorMake(40.0*CGFloat(value!), 0)
    }
}
```

```
self.playerShip.physicsBody?.applyForce(fvector)
}
```

⚠注意：如果你的设备是用屏幕朝上和按键的主页键来调整方向的，那么向右倾斜设备会产生 $\text{value} > 0$ ，而向左倾斜设备会产生 $\text{value} < 0$ 。这个检查结果 0.2 意味着设备被认为是完全平放的。

此时运行程序，可以看到以下的运行效果。当设备向左倾斜时，可以看到飞船向左进行移动，效果如图 10.20 所示。

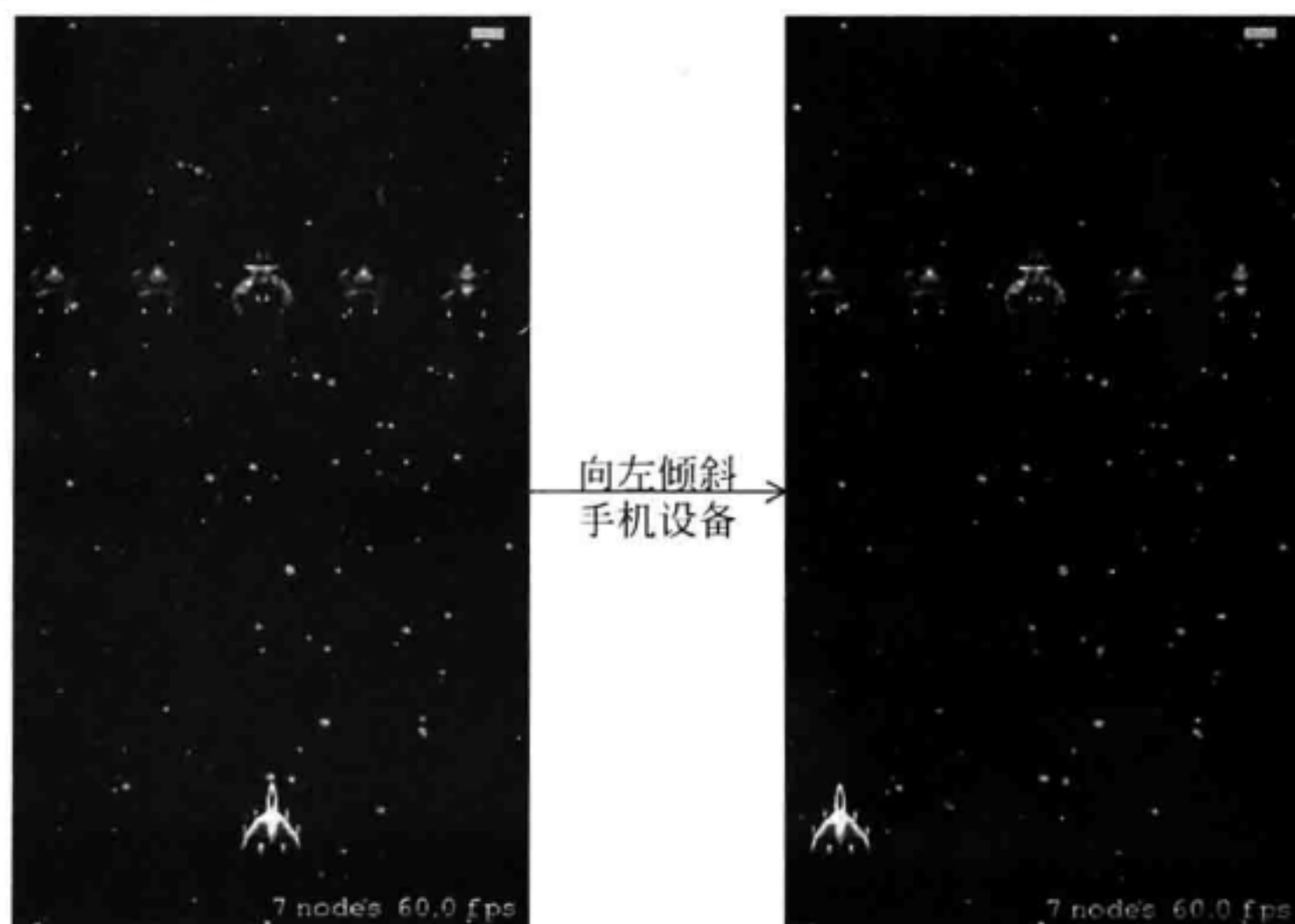


图 10.20 运行效果

当设备向右倾斜时，可以看到飞船向右进行移动，效果如图 10.21 所示。

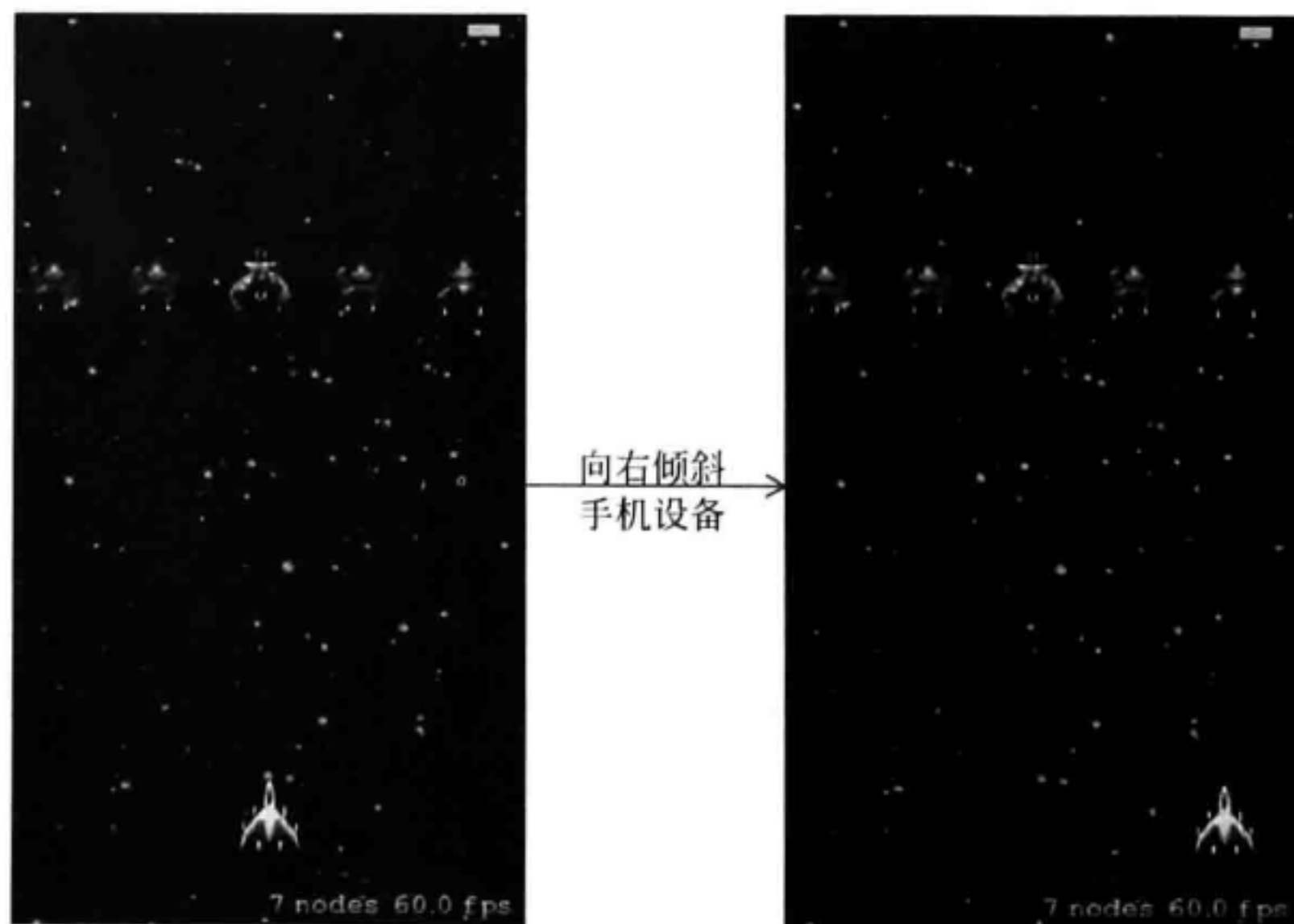


图 10.21 运行效果

⚠注意：此游戏需要使用真机进行测试，因为在模拟器中没有传感器。

10.8 碰撞检测

飞船的子弹是用来消灭敌人的，如何实现这一功能，就是本节将要讲解的内容。子弹消灭敌人的实现思路其实很简单，就是在玩家发射子弹后，判断子弹是否和某一敌人相交，如果相交，就隐藏此敌人，表明敌人被消灭了。具体的实现步骤如下所述。

(1) 在 `update(currentTime:CFTimeInterval)` 方法中添加以下代码，实现 `collisionDetection()` 方法的调用：

```
self.shipUpdates()  
self.collisionDetection()
```

(2) 添加 `collisionDetection()` 方法，在此方法中编写代码，实现子弹击中敌人的功能，代码如下：

```
func collisionDetection() {  
    //实例化两个:SKSpriteNode 对象  
    var enemy:SKSpriteNode=SKSpriteNode()  
    var senemy:SKSpriteNode=SKSpriteNode()  
    //遍历数组 enemies  
    for enemy in self.enemies{  
        senemy=enemy as SKSpriteNode  
        //判断 senemy 对象是否隐藏  
        if(senemy.hidden==true){  
            continue  
        }  
        //实例化两个:SKSpriteNode 对象  
        var bullet:SKSpriteNode=SKSpriteNode()  
        var sbullet:SKSpriteNode=SKSpriteNode()  
        //遍历数组  
        for bullet in self.playerBullets {  
            sbullet=bullet as SKSpriteNode  
            //判断 sbullet 是否隐藏  
            if(sbullet.hidden==true){  
                continue  
            }  
            //判断 bullet 节点和 enemy 节点是否相交  
            if(bullet.intersectsNode(enemy as SKNode)){  
                sbullet.hidden=false  
                senemy.hidden=true  
                continue  
            }  
        }  
    }  
}
```

此时运行程序，会看到如图 10.22 所示的效果。

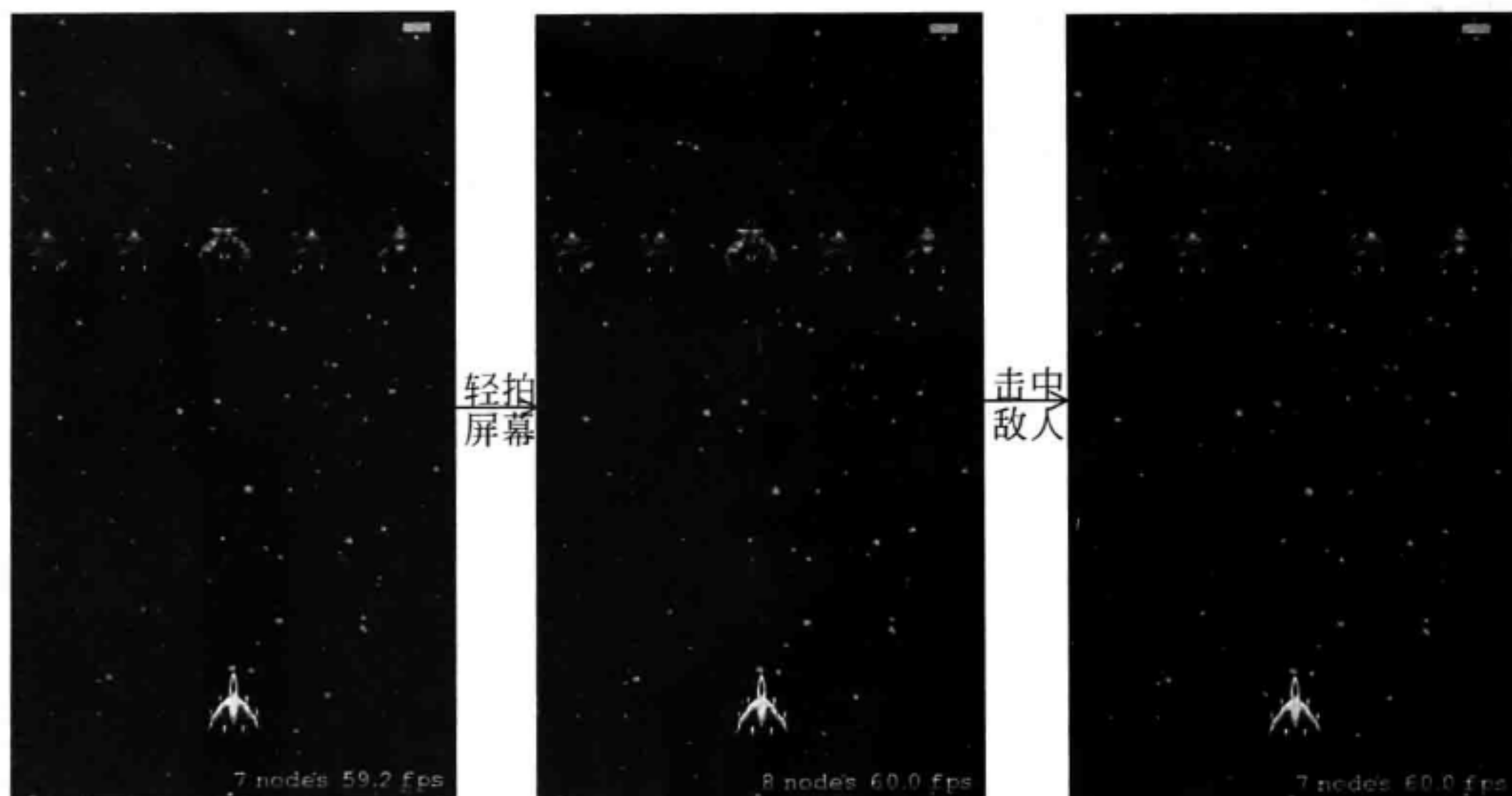



图 10.22 运行效果

 **注意：**在此代码中使用到了 `intersectsNode(node:SKNode)` 方法。此方法可以用来判断一个节点是否与另外一个节点相交。如果返回值为 `true`，则表明这两个节点相交；如果返回值为 `false`，则表明这两个节点不相交。

10.9 分数显示

在本书的每一个游戏中都会存在一个类似于分数显示的功能。对于本章中的游戏也不例外。本节将讲解分数的显示。

10.9.1 使用 SKLabelNode 添加显示分数的节点

分数一般是在标签中进行显示的。但在 Sprite Kit 技术中，需要在 SKLabelNode 产生的对象中进行显示。在显示分数前，首先需要添加一个 SKLabelNode 对象的节点，添加方式类似于背景、飞船等的添加。具体的实现步骤如下所述。

(1) 添加两个变量，代码如下：

```
var scoreLabel:SKLabelNode=SKLabelNode()           //用来显示分数的节点
var playerScore:Int=0                               //用来保存分数
```

(2) 在 `didMoveToView(view: SKView)` 方法中添加以下的代码，实现 SKLabelNode 对象的设置以及添加。

```
self.playerShip.physicsBody?.mass=0.2
self.scoreLabel = SKLabelNode(fontNamed:"Chalkduster")
self.scoreLabel.fontSize=30                        //设置字体的大小
```

```

self.scoreLabel.horizontalAlignmentMode=SKLabelHorizontalAlignmentMode.
Left           //设置对齐方式
self.scoreLabel.position=CGPointMake(30,CGRectGetMaxY(self.frame)-50)
               //设置位置
self.scoreLabel.text="\ (playerScore) "                //设置显示内容
self.addChild(self.scoreLabel)

```

此时运行程序，会看到如图 10.23 所示的效果。

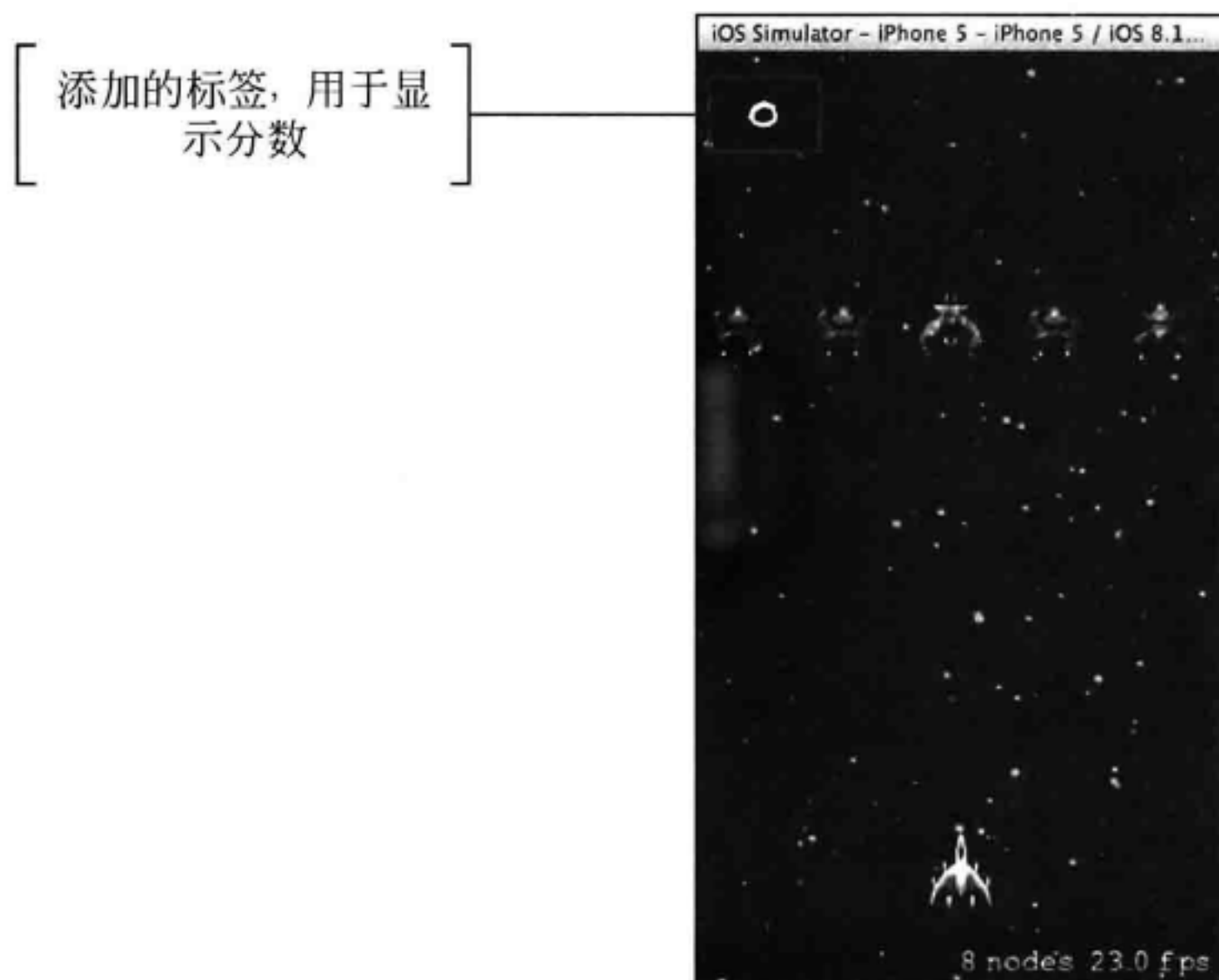


图 10.23 运行效果

10.9.2 实现分数的显示

分数的显示其实很简单，就是在每消灭一个敌人后，将 playerScore 自加 1，所以需要在 collisionDetection() 方法中添加以下的代码：

```

for enemy in self.enemies{
    .....
    for bullet in self.playerBullets {
        .....
        if(bullet.intersectsNode(enemy as SKNode)){
            sbullet.hidden=false
            senemy.hidden=true
            playerScore++
            self.scoreLabel.text="\ (playerScore) "
            continue
        }
    }
}

```

此时运行程序，就会看到如图 10.24 所示的效果。

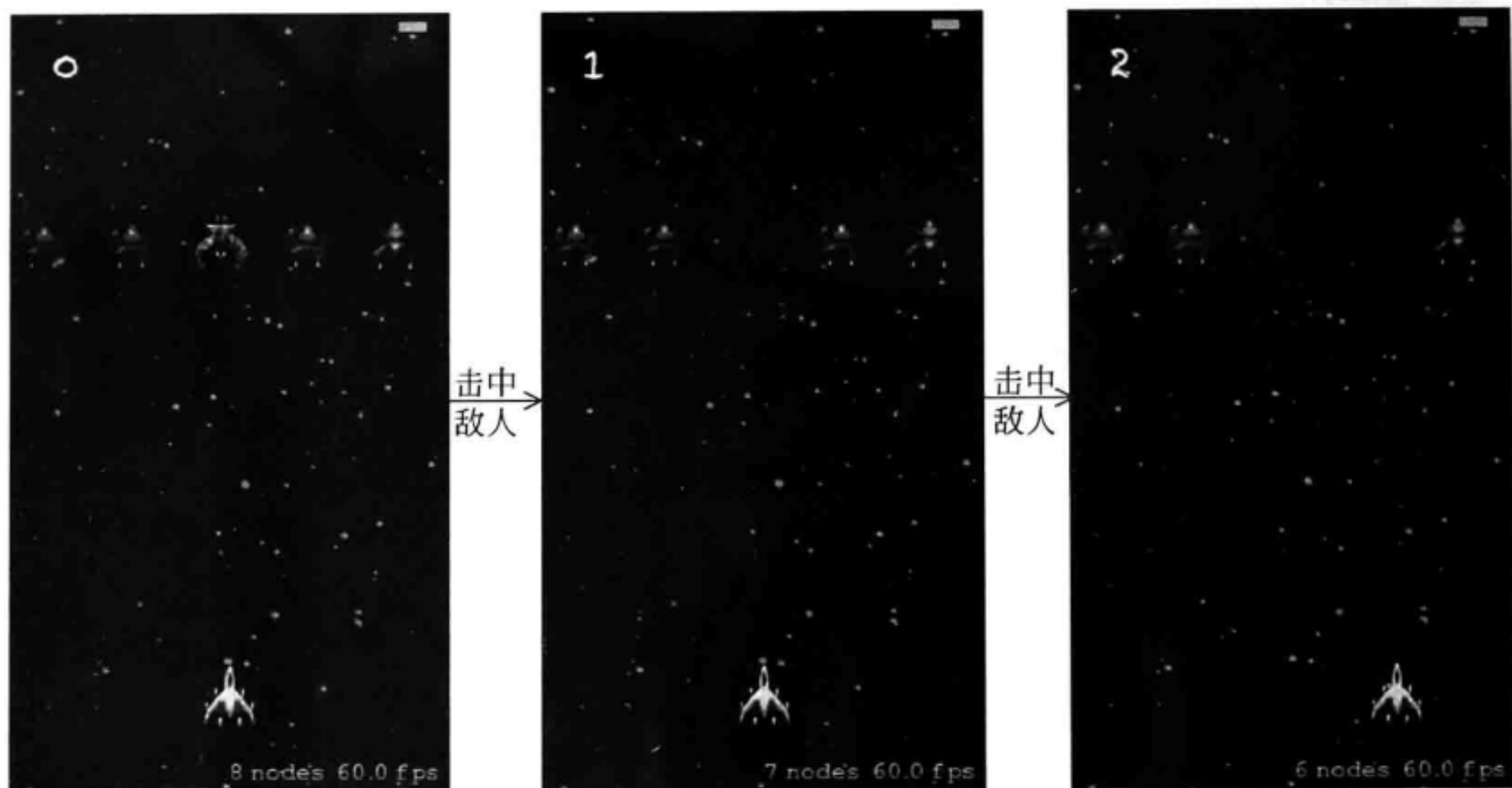


图 10.24 运行效果

10.10 添加声音

为了让此游戏更具真实感，我们还需要在游戏中添加声音，起到一个提示的作用。

10.10.1 进入射击游戏界面的声音

首先在刚进入射击游戏界面时添加一个声音，此声音用来提醒玩家，游戏开始了。以下就是添加声音的具体实现方式。

(1) 导入 AVFoundation 框架，代码如下：

```
import AVFoundation
```

(2) 创建一个用来播放音频文件的音频播放器对象，代码如下：

```
var audioEffect:AVAudioPlayer?=nil
```

此声音由于是在刚进入射击游戏界面时播放的，所以需要在 `didMoveToView(view: SKView)` 方法中编写以下的代码：

```
self.addChild(self.scoreLabel)
var soundName:NSString = "startup"
self.playSoundEffect(soundName)
```

(3) 在 `playSoundEffect(effectName:NSString)` 方法中实现声音的播放，代码如下：

```
func playSoundEffect(effectName:NSString) {
    var path=NSBundle.mainBundle().pathForResource(effectName, ofType: "wav")
    var pathURL=NSURL.fileURLWithPath(path!)
```



```

audioEffect=AVAudioPlayer(contentsOfURL: pathURL,error: nil)
audioEffect?.numberOfLoops=0
audioEffect?.prepareToPlay()
audioEffect?.play()
}

```

此时运行程序，当界面切换到射击游戏界面时，就会播放 startup.wav 音频文件的声音了。

10.10.2 子弹击中敌人的声音

子弹击中敌人添加的声音是用来更清楚的提醒玩家，已成功歼灭敌人。所以需要在 collisionDetection() 方法中添加以下的代码：

```

if(bullet.intersectsNode(enemy as SKNode)){
    .....
    self.scoreLabel.text="\ (playerScore) "
    var soundName:NSString="impact"
    self.playSoundEffect(soundName)
    continue
}

```

此时运行程序，玩家在触摸屏幕后发出子弹。如果发出的子弹击中敌人，就会播放 impact.wav 音频文件中的声音。如果发出的子弹没有击中敌人，impact.wav 音频文件中的声音就不会进行播放。

10.11 游戏介绍模块

在游戏介绍模块的界面中提供了对此游戏的介绍以及玩法。本节将讲解关于游戏介绍模块中对界面的设计。在视图对象库中拖动 View Controller 视图控制器对象到画布中，然后对此视图控制器的界面进行设计，效果如图 10.25 所示。

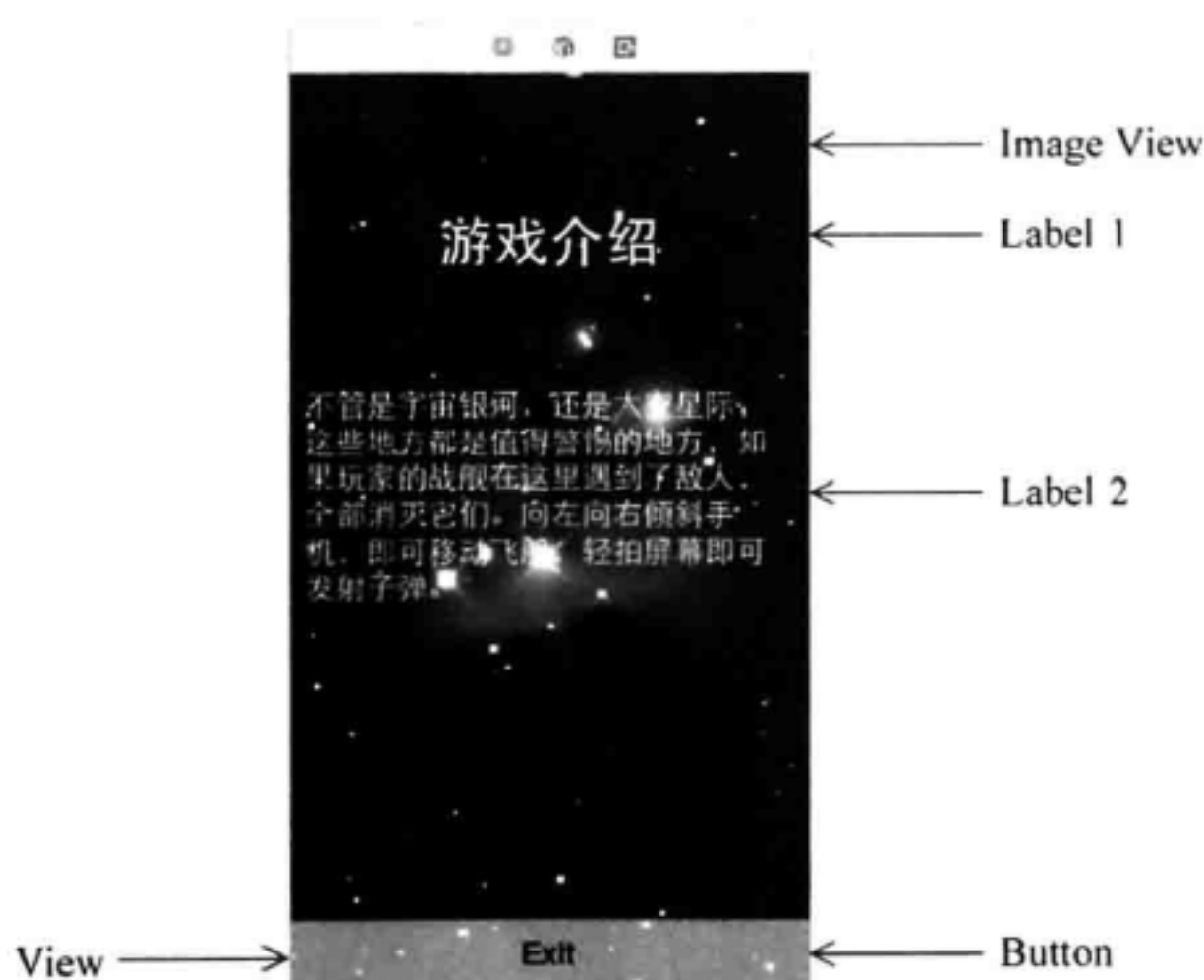


图 10.25 界面效果

需要添加的视图对象，以及对它们的设置，如表 10-4 所示。

表 10-4 设置界面

视 图	设 置
Image View	Image: background1.png 位置和大小: (0, 0, 320, 568)
Label1	Text: 游戏介绍 Color: 白色 Font: System Bold 34.0 Alignment: 居中 位置和大小: (90, 84, 140, 41)
Label2	Text: 不管是宇宙银河，还是太空星际，这些地方都是值得警惕的地方，如果玩家的战舰在这里遇到了敌人，全部消灭它们。向左向右倾斜手机，即可移动飞船，轻拍屏幕即可发射子弹。 Color: 白色 Font: System 19.0 Lines: 6 位置和大小: (9, 156, 303, 212)
View	Alpha: 0.6 位置和大小: (0, 525, 320, 43)
Button	Title: Exit Font: System Bold 19.0 Text Color: 黑色 位置和大小: (137, 7, 46, 30)


10.12 场 景 切 换

在此游戏中需要将所有的场景进行切换，具体的切换关系如表 10-5 所示。

表 10-5 场景切换

对 象	切 换 到
Play Game 按钮	射击游戏界面
Game Description 按钮	游戏介绍界面
射击游戏界面中的 Exit 按钮	主菜单界面
游戏介绍界面中的 Exit 按钮	

最后画布中的效果如图 10.26 所示。

 **注意：**在实现场景切换前，首先需要单击实现主菜单的视图控制器，在此视图控制器中，选择界面上方的 Dock 中的 View Controller 图标，在工具窗口中的 Show the Attributes inspector 选项，即属性查看器选中，找到 Is Initial View Controller 复选框，将其选中，使此视图控制器成为初始视图控制器。

此时运行程序，可以看到以下的效果。当玩家在主菜单中轻拍 Play Game 按钮，可以进入射击游戏界面。当玩家轻拍 Exit 按钮，可以返回到主菜单，效果如图 10.27 所示。

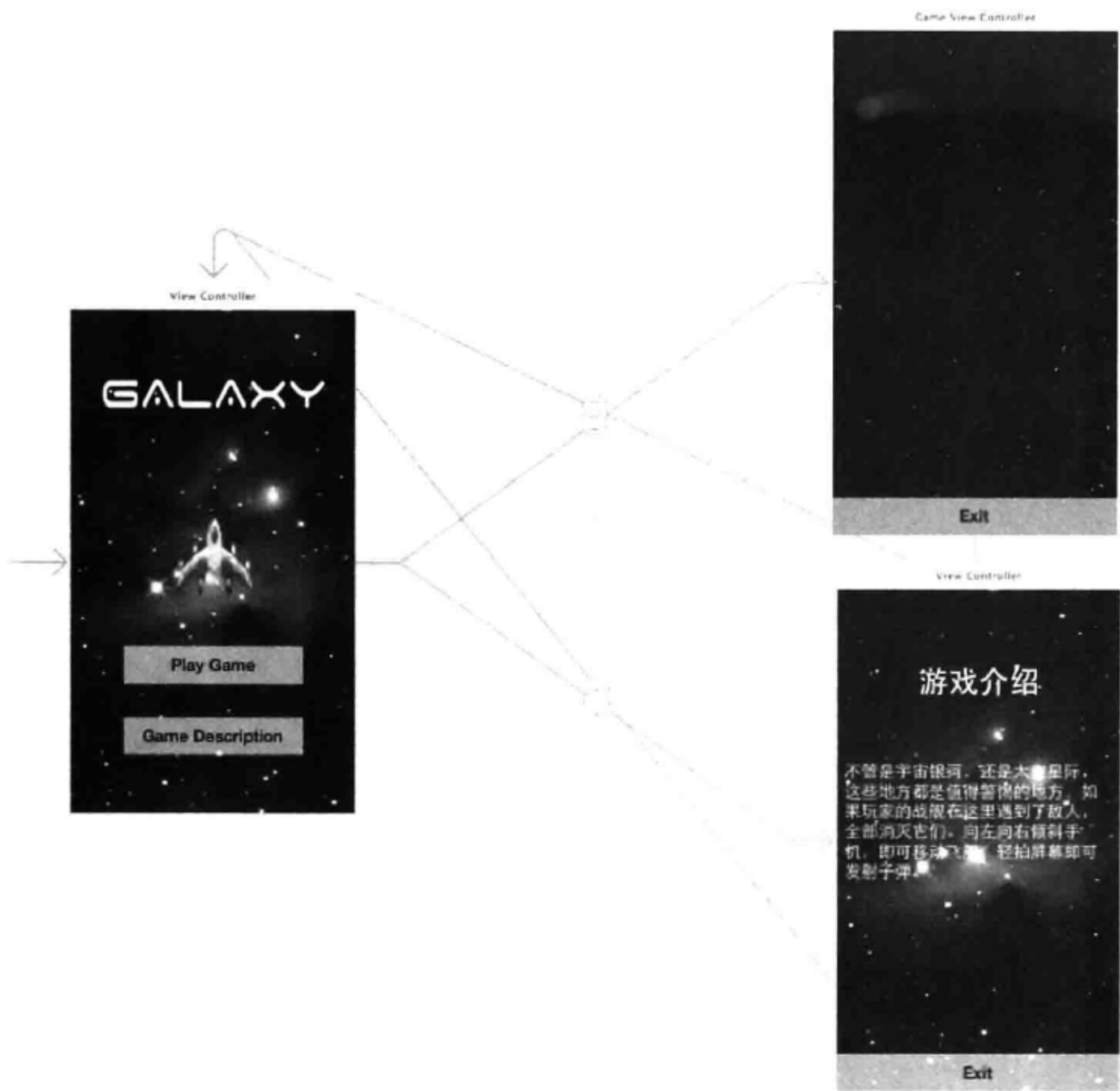


图 10.26 画布的效果

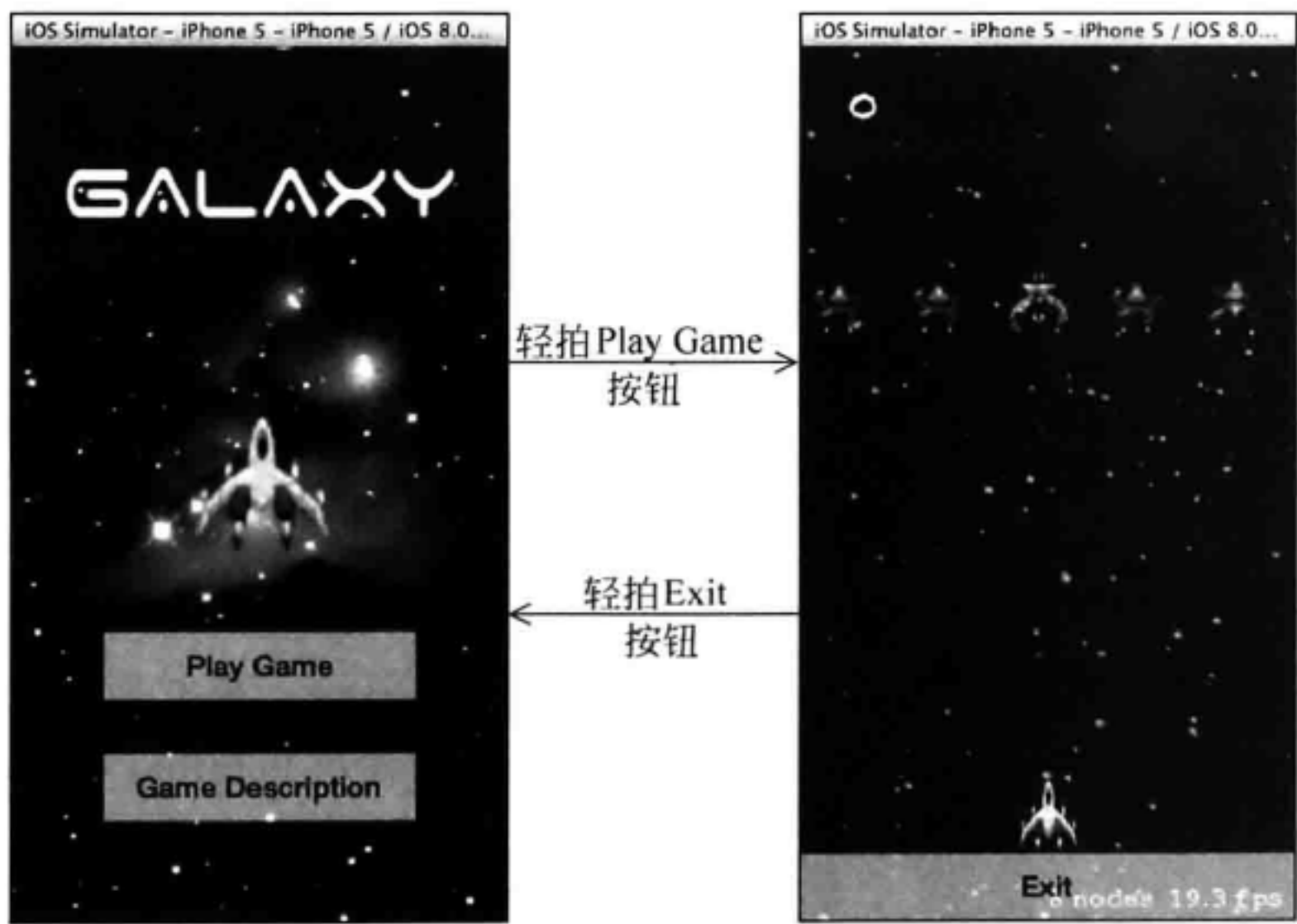


图 10.27 运行效果 1

当玩家在主菜单中轻拍 Game Description 按钮，可以进入游戏介绍界面。当玩家轻拍 Exit 按钮，可以返回到主菜单，效果如图 10.28 所示。



图 10.28 运行效果 2

第 11 章 应用程序的发布

开发者开发的应用程序一般都需要上传到 App Store 上才可以被 iOS 的用户看到。如果上传的软件是收费的，那么当用户下载后，开发者就可以从中得到一定的利益。App Store 是 Apple 推出的在线商店。本章以开发的 Simon Memory Game 游戏为例，来为开发者讲解应用程序的发布。

11.1 创建 App ID

在发布应用程序之前，首先需要做的就是创建一个 App ID，每一个被发布的应用程序都必须有一个唯一标识的 App ID。在这里，我们创建的 App ID 为 MySimonGame，Bundle ID 为 com.MySimonGame.app。具体步骤如下所示。

(1) 在 Safari 的搜索栏中输入网址（<https://developer.apple.com/devcenter/ios/index.action>），然后按回车键，进入 iOS Dev Center-App Developer 网页。

(2) 单击 Log in 按钮，进入 Sign in with your Apple ID-Apple Developer 网页。在此网页中需要开发者输入 Apple ID 以及密码。然后单击 Sign In 按钮，此时会再次进入 iOS Dev Center-App Developer 网页，如图 11.1 所示。

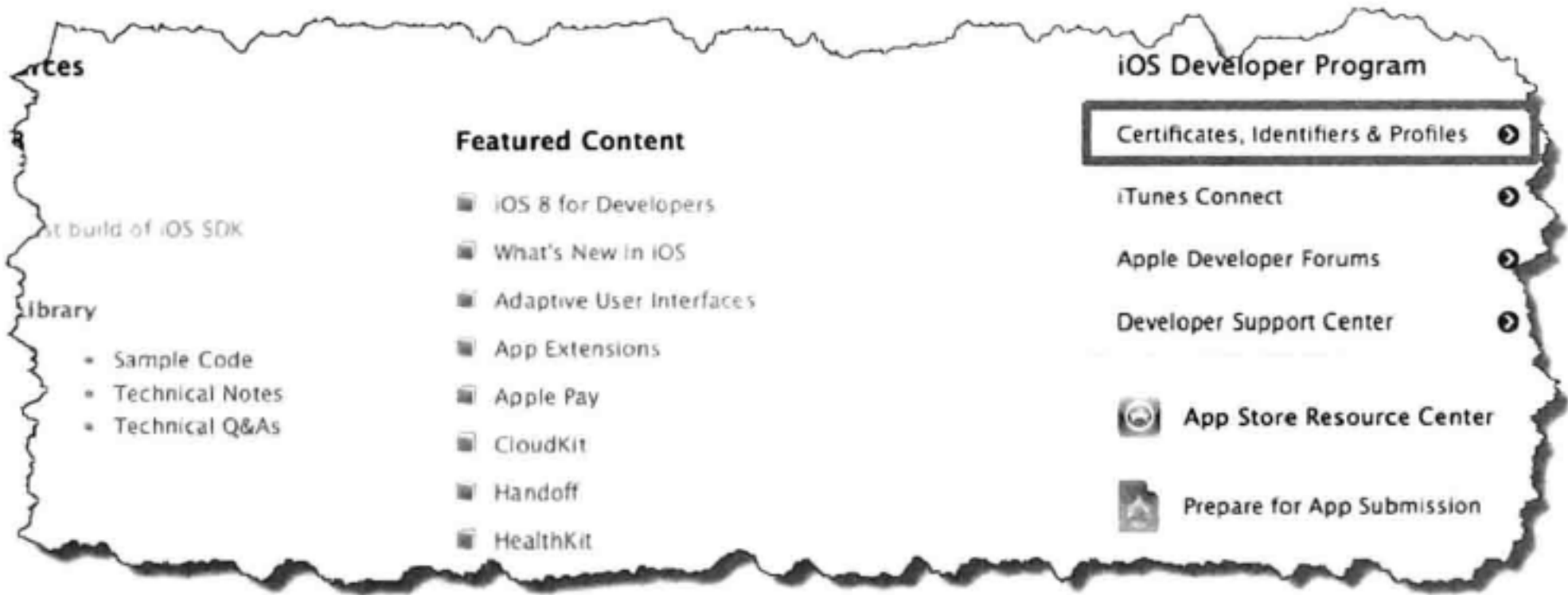


图 11.1 操作步骤

(3) 选择 Certificates,Identifiers&Profiles 选项，进入 Certificates,Identifiers &Profiles-App Developer 网页，如图 11.2 所示。

(4) 选择 Identifiers 选项，进入 iOS App IDs-Apple Developer 网页，实现一个 App ID 的创建，由于在前面讲解过 App ID 的创建，这里就不进行详细的介绍了，需要注意的是，在 App Services 中，不需要选择 Associate Domains 选项，如图 11.3 所示。



图 11.2 操作步骤

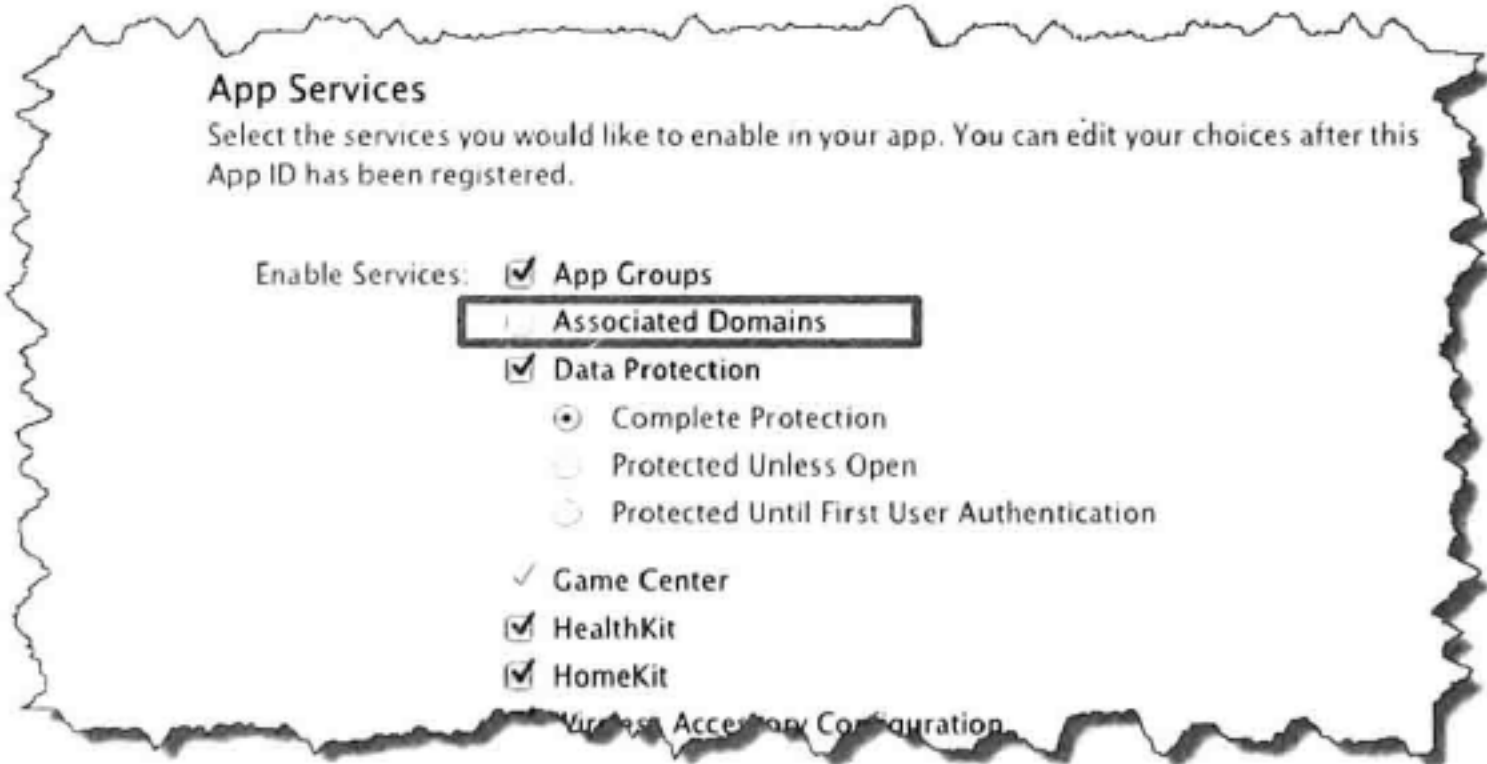


图 11.3 App Services

11.2 申请发布证书

创建好 App ID 后，还需要申请发布证书。本节将讲解如何申请发布证书。

11.2.1 申请证书

以下是申请发布证书的具体步骤。

(1) 如果开发者还处于创建 App ID 的网页中，可以选择此网页右侧的 Certificates 的 Production 选项，进入 iOS Certificates (Production)-Apple Developer 网页，如图 11.4 所示。

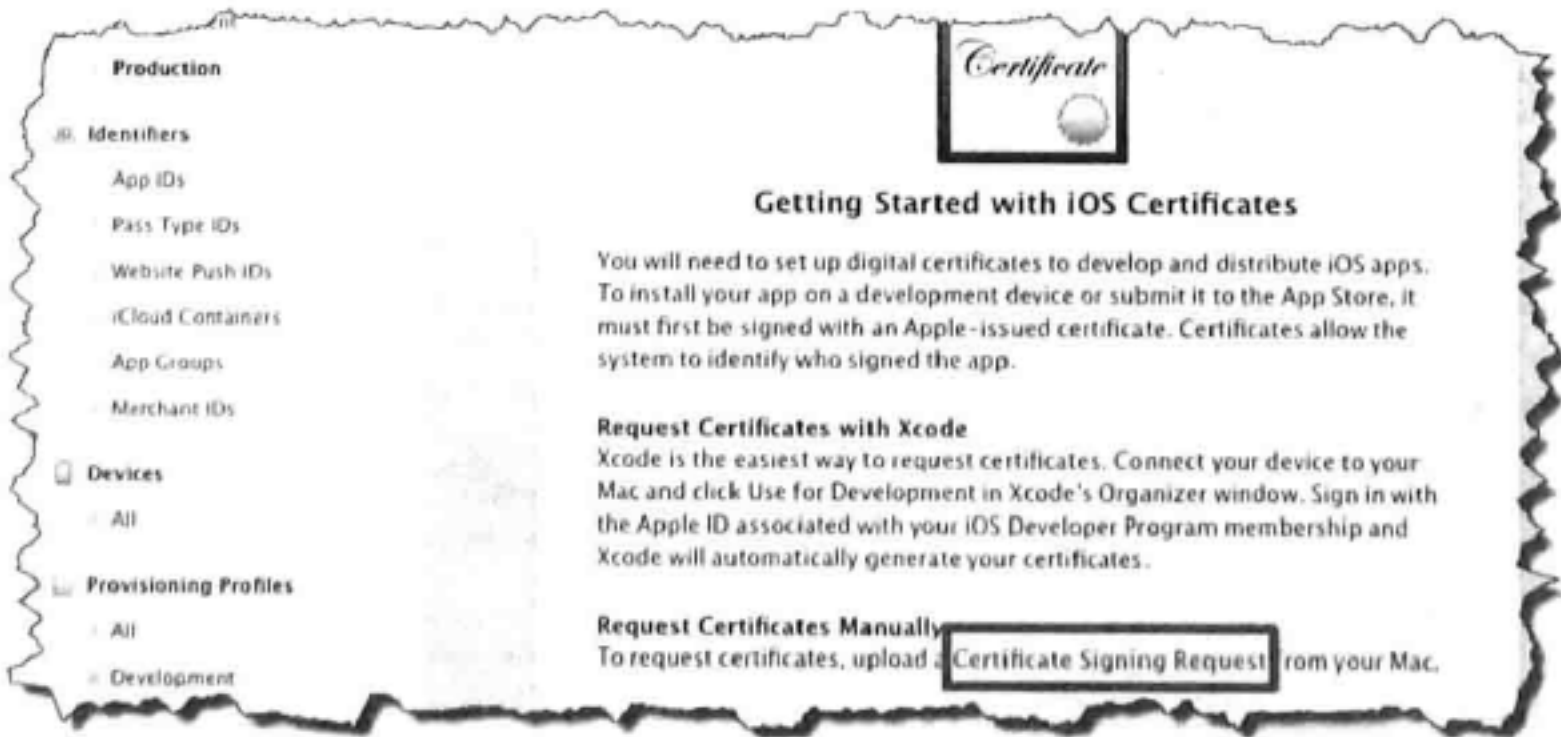


图 11.4 操作步骤 1

(2) 在此网页中，选择蓝色的 Certificate Signing Request 字符串，进入 Add-iOS Certificates-AppleDeveloper 网页，如图 11.5 所示。



图 11.5 操作步骤 2

(3) 选择 App Store and Ad Hoc 单选按钮。然后单击 Continue 按钮，进入 Request 选项卡的网页中，如图 11.6 所示。

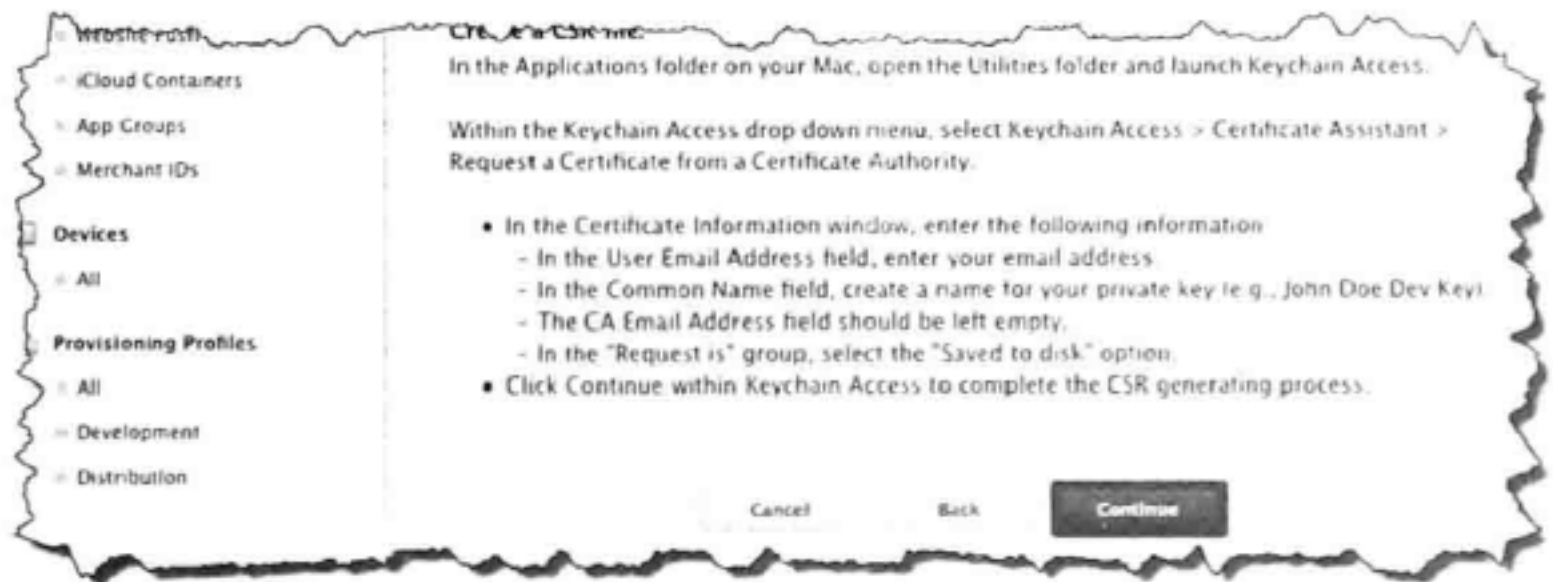


图 11.6 操作步骤 3

(4) 单击 Continue 按钮，进入 Generate 选项卡的网页中，如图 11.7 所示。



图 11.7 操作步骤 4

(5) 选择 Choose File...按钮后，弹出选择文件对话框，如图 11.8 所示。



图 11.8 操作步骤 5

(6) 选择在桌面的 CertificateSigningRequest.certSigningRequest 文件, 此文件就是生成的证书签名申请(在第 1 章中讲解过此文件的申请), 然后单击“选取”按钮。再单击 Generate 按钮, 进入 Download 选项卡的网页中, 如图 11.9 所示。



图 11.9 操作步骤 6

(7) 单击 Download 按钮, 对生成的证书进行下载。下载的后的证书名为 ios_distribution.cer。

(8) 双击下载的 ios_distribution.cer 证书, 将此证书添加到钥匙串中。

11.2.2 申请证书对应的配置文件 (Provision File)

以下是申请发布证书对应的配置文件的具体步骤。

(1) 如果开发者还处于下载证书的网页, 可以选择此网页右侧的 Provisioning Profiles 的 Distribution 选项, 进入 iOS Provisioning Profiles (Distribution)-Apple Developer 网页, 如图 11.10 所示。

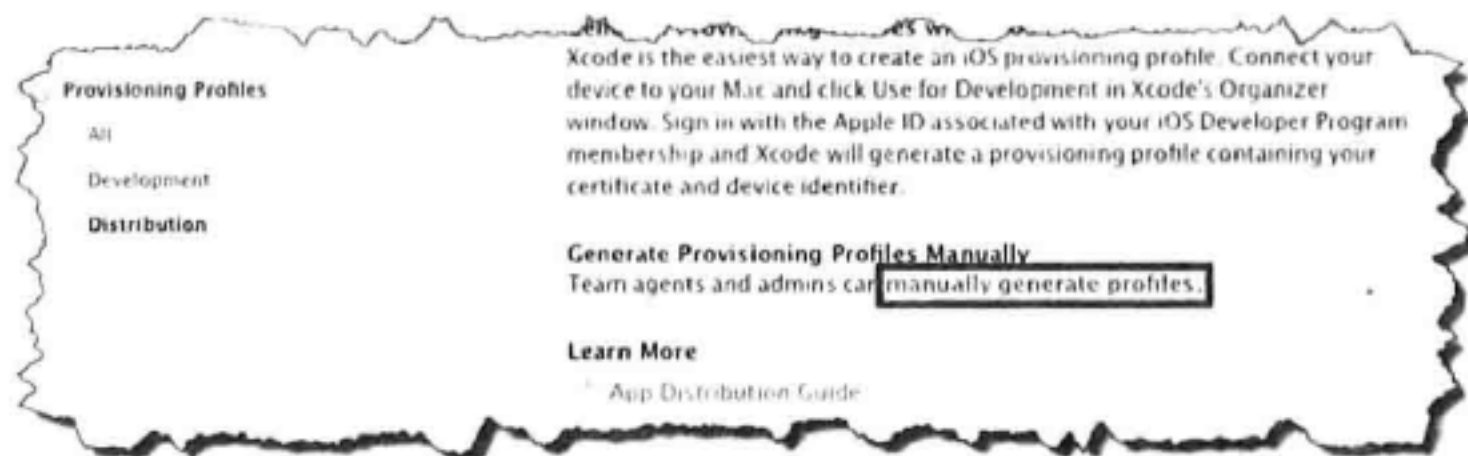


图 11.10 操作步骤 1

(2) 选择蓝色的 manually generate profiles 字符串, 进入 Add-iOS Provisioning Profile-Apple Developer 网页, 如图 11.11 所示。

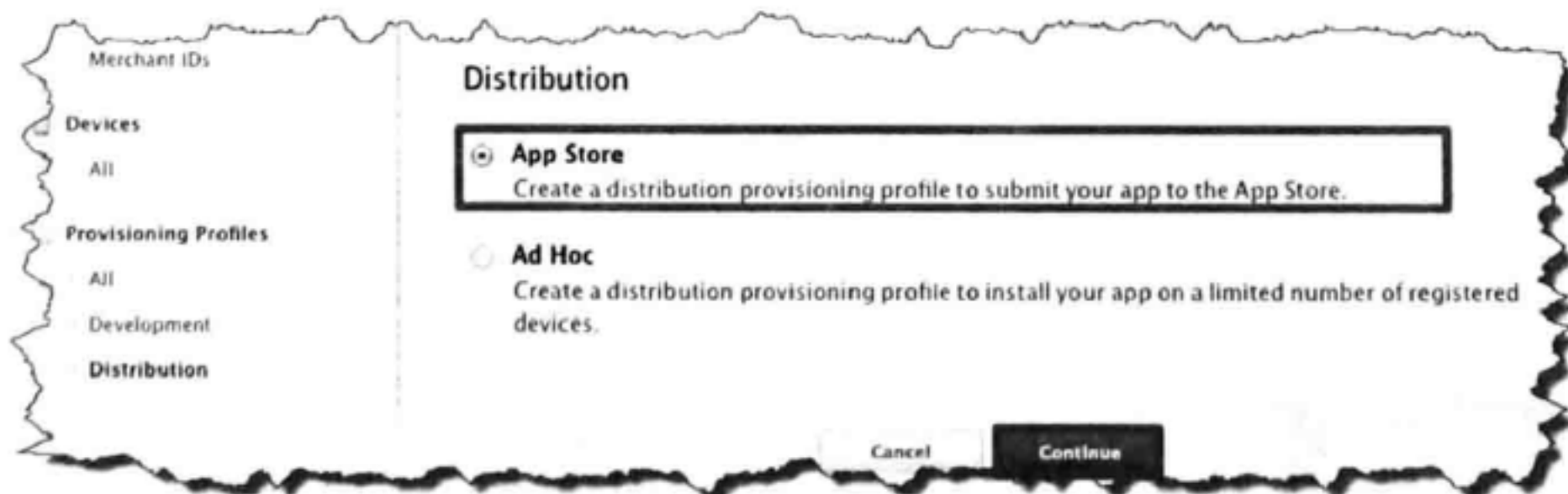


图 11.11 操作步骤 2

(3) 选择 App Store, 然后单击 Continue 按钮, 进入 Configure 选项卡的网页中, 如图 11.12 所示。



图 11.12 操作步骤 3

(4) 选择 App ID, 然后单击 Continue 按钮, 进入 Configure 选项卡的选择证书的网页中, 如图 11.13 所示。



图 11.13 操作步骤 4

(5) 选择某一个证书单选按钮, 然后单击 Continue 按钮, 进入 Generate 选项卡的网页中, 如图 11.14 所示。

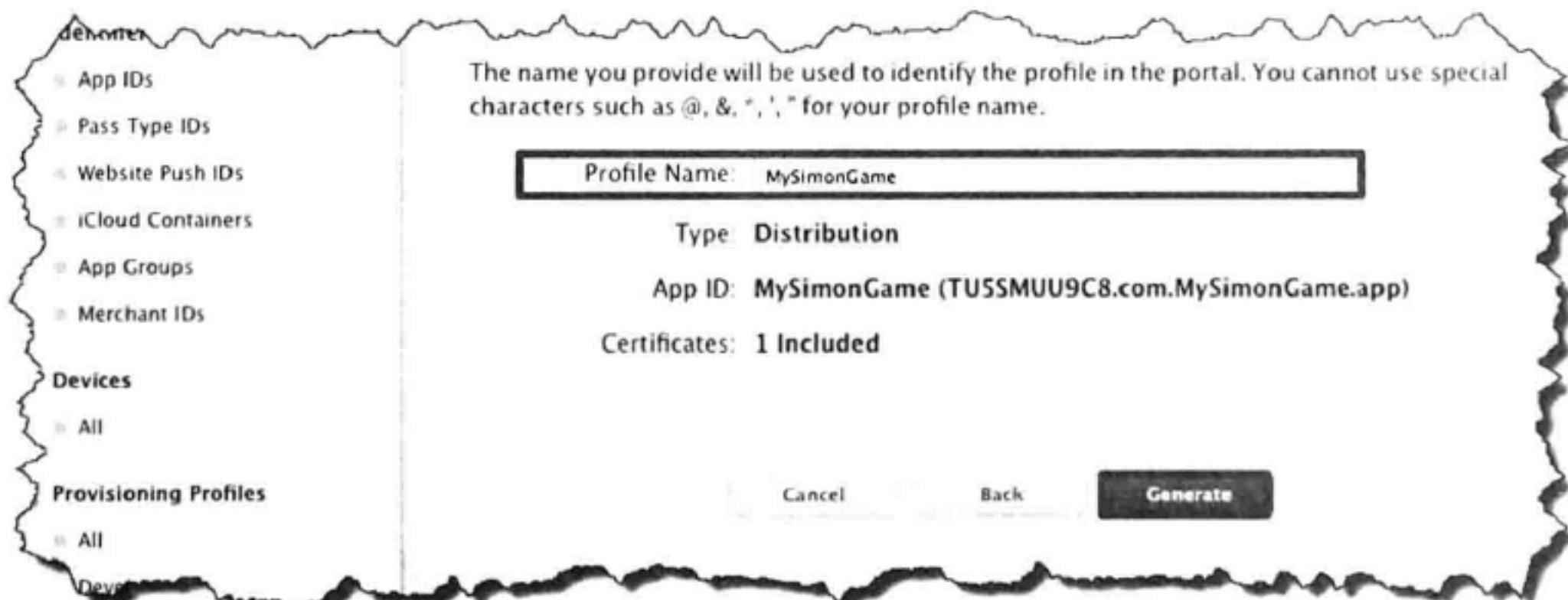


图 11.14 操作步骤 5

(6) 输入配置的文件名, 然后单击 **Generate** 按钮, 进入 **Download** 选项卡的网页中, 如图 11.15 所示。



图 11.15 操作步骤 6

(7) 选择 **Download** 按钮, 对 **Provisioning Profiles** 进行下载, 下载后的文件为 **MySimonGame.mobileprovision**。

(8) 双击下载的 **MySimonGame.mobileprovision** 文件, 将此文件添加到 **Organizer** 的 **Provisioning Profiles** 中。

11.3 准备提交应用程序

在提交应用程序之前, 需要做一些准备工作。否则, 你的应用程序是无法提交的。本节将针对这些准备工作进行讲解。

11.3.1 创建应用及基本信息

要提交一个应用程序, 首先需要在 **iTunes Connect** 的网页中对这个应用程序进行创建和填写一些基本信息。以下是它的具体操作步骤。

(1) 在 **Safari** 的搜索栏中输入网址 (<http://itunesconnect.apple.com>), 然后按回车键, 进入 **iTunes Connect** 的登录网页, 如图 11.16 所示。

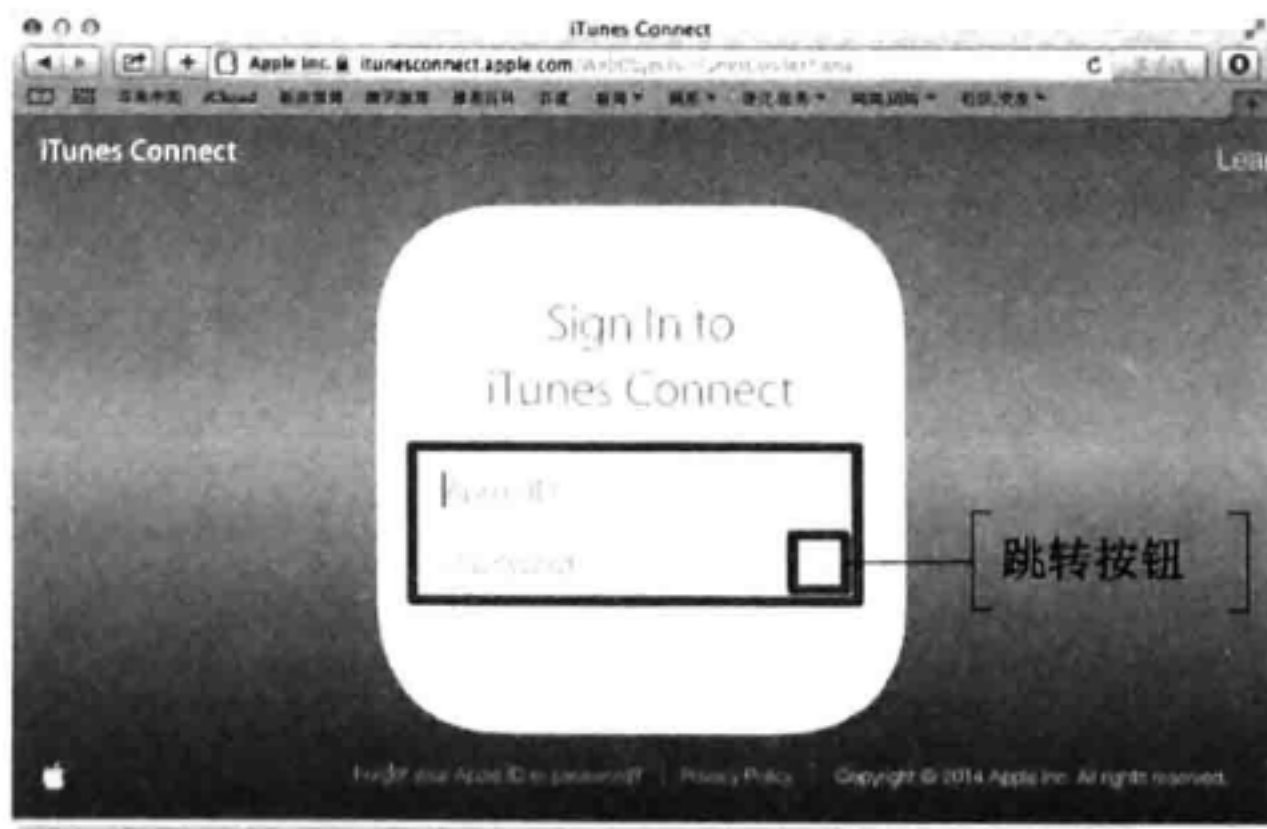



图 11.16 操作步骤 1

(2) 输入苹果账号和密码后，单击跳转按钮进入服务条款的网页，如图 11.17 所示。



图 11.17 操作步骤 2

 **注意：**如果开发者是第一次使用 iTunes Connect，才会弹出如图 11.17 所示的页面，如果不是，就不会弹出此页面。

(3) 选择 I have read and agree to the above Terms of Service 复选框，然后触摸 Accept 按钮。进入 iTunes Connect 的网页，如图 11.18 所示。

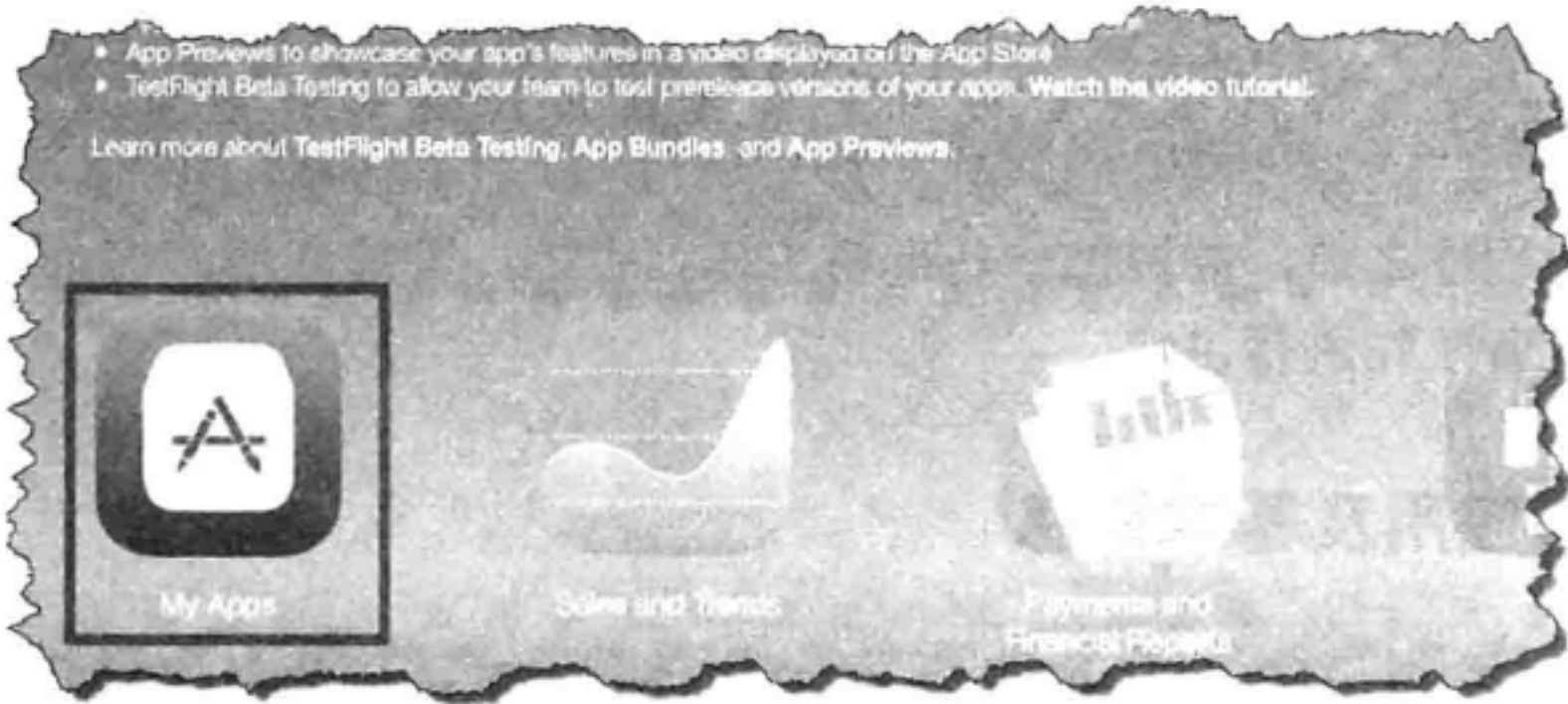


图 11.18 操作步骤 3

(4) 选择 My Apps，进入 My Apps 网页，在这里面放置了一些上传的应用程序，如果你是第一次使用，就是空的，如图 11.19 所示。



图 11.19 操作步骤 4

(5) 单击+按钮，弹出下拉菜单，如图 11.20 所示。

注意：如果在你的 My Apps 中存在一些应用程序，那么在单击+按钮后，出现的下拉菜单如图 11.21 所示。选择其中的 New iOS App 选项创建一个新的 iOS 应用程序。



图 11.20 操作步骤 5



图 11.21 下拉列表

(6) 在弹出的下拉菜单中选择 New iOS App 选项，弹出 New iOS App 对话框，在此对话框中输入相应的内容，如图 11.22 所示。

New iOS App

Name

MySimonGame

Version

1.0

Primary Language

English

SKU

com.MySimonGame.app

Bundle ID


MySimonGame - com.MySimonGame.app

Remember, create a unique ID on the iTunes Connect site.

Cancel

Create

图 11.22 操作步骤 6

注意：App Name 必须是 App Store 未使用的，当填入的时候，系统会检查。在 Bundle ID 中输入应用程序标识符，它是在 iOS 开发中心的配置门户创建 App ID 时生成的。如果在配置门户网站中有，就可以在下拉列表中找到。SKU 是应用程序编号，具有唯一性。

(7) 单击 Create 按钮，进入此应用的详细信息的网页，如图 11.23 所示。开发者需要在此网页中输入一些信息。

11.3.2 项目的相关设置

一个应用程序在编程过程中，属性并不影响开发，即使这些属性设置的是错误的。但是在发布时，正确的设置这些属性是很重要的。以下就是对这些属性的设置。

1. 设置标识符

标识符在开发过程中对我们来说并没有什么影响，但是在发布时非常重要。打开项目

的目标窗口，选择 **General** 选项，在其中找到 **Bundle Identifier**，将它设置为 `com.MySimonGame.app`，如图 11.24 所示。



图 11.23 操作步骤 7

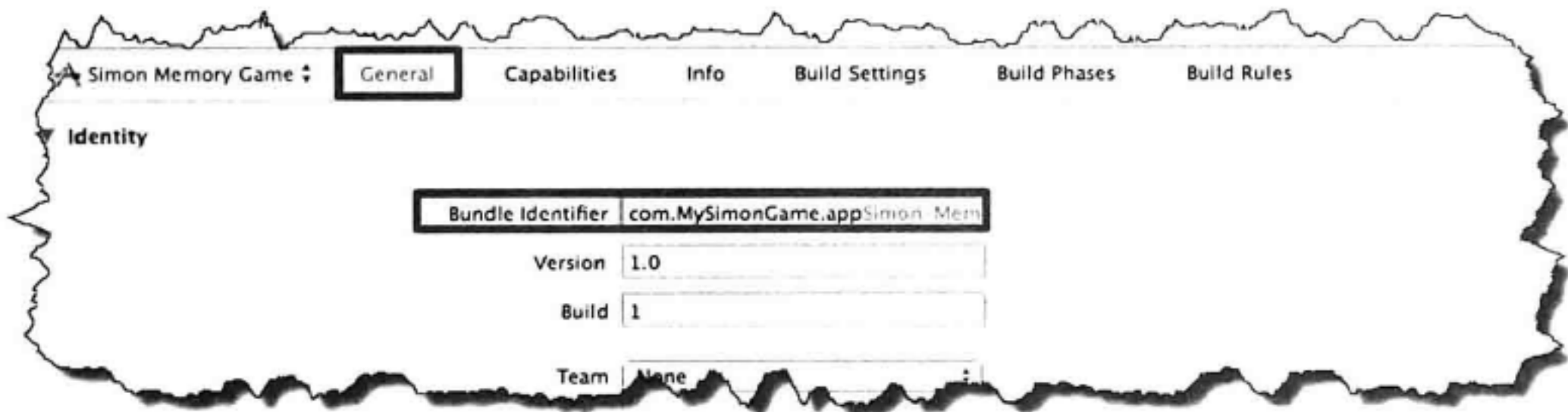


图 11.24 设置标识符

注意：在 **Bundle Identifier** 中可以看到，在标识符 `com.MySimonGame.app` 的后面还有一些内容，这些内容是灰色的，表示它是不可以进行操作的。我们需要选择 **Info** 选项，在其中找到 **Bundle Identifier**，将此项中的内容改为 `com.MySimonGame.app`，如图 11.25 所示。此时再回到 **General** 选项中，就可以看到 **Bundle Identifier** 中没有灰色的这部分内容了。

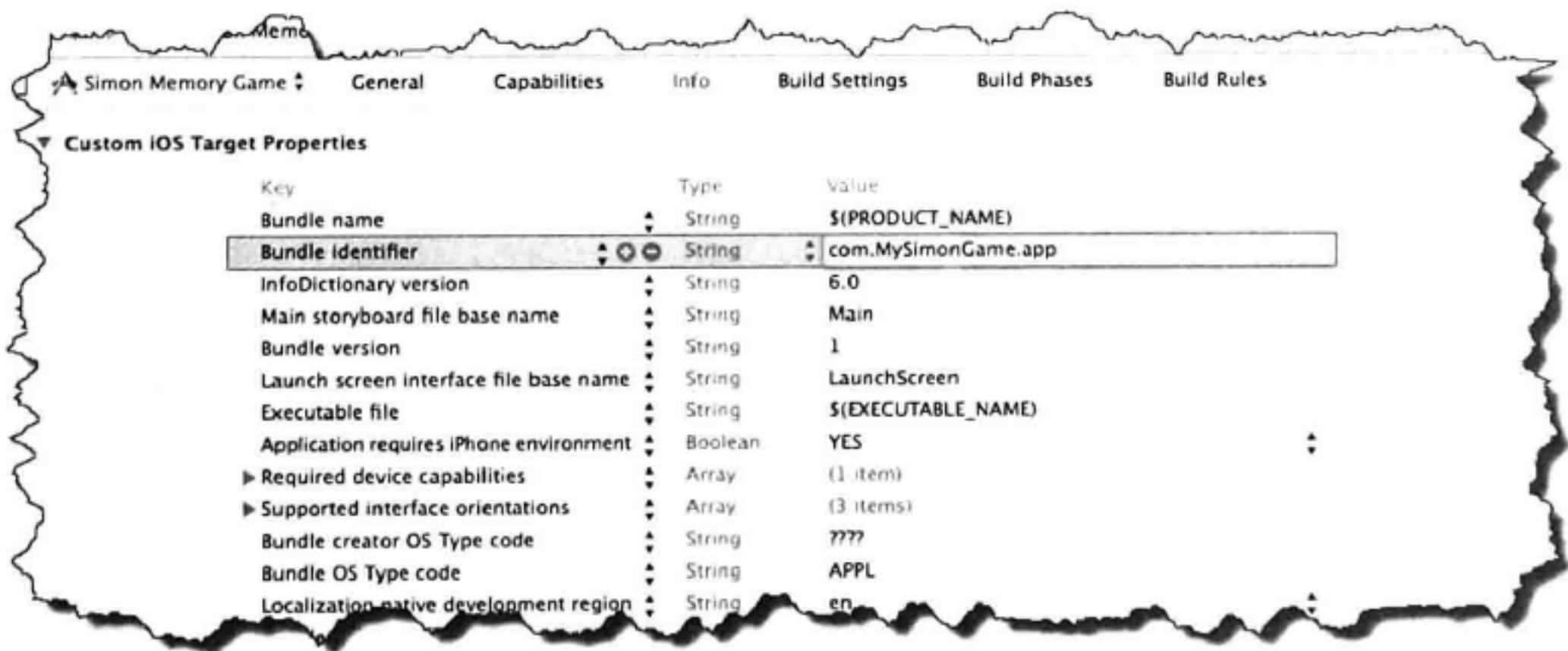


图 11.25 Info

2. 设置图标

每一个应用程序都必须要有图标，否则发布的应用程序是通不过审核的，图标的设置在前面已经讲解过了，如图 11.26 所示。

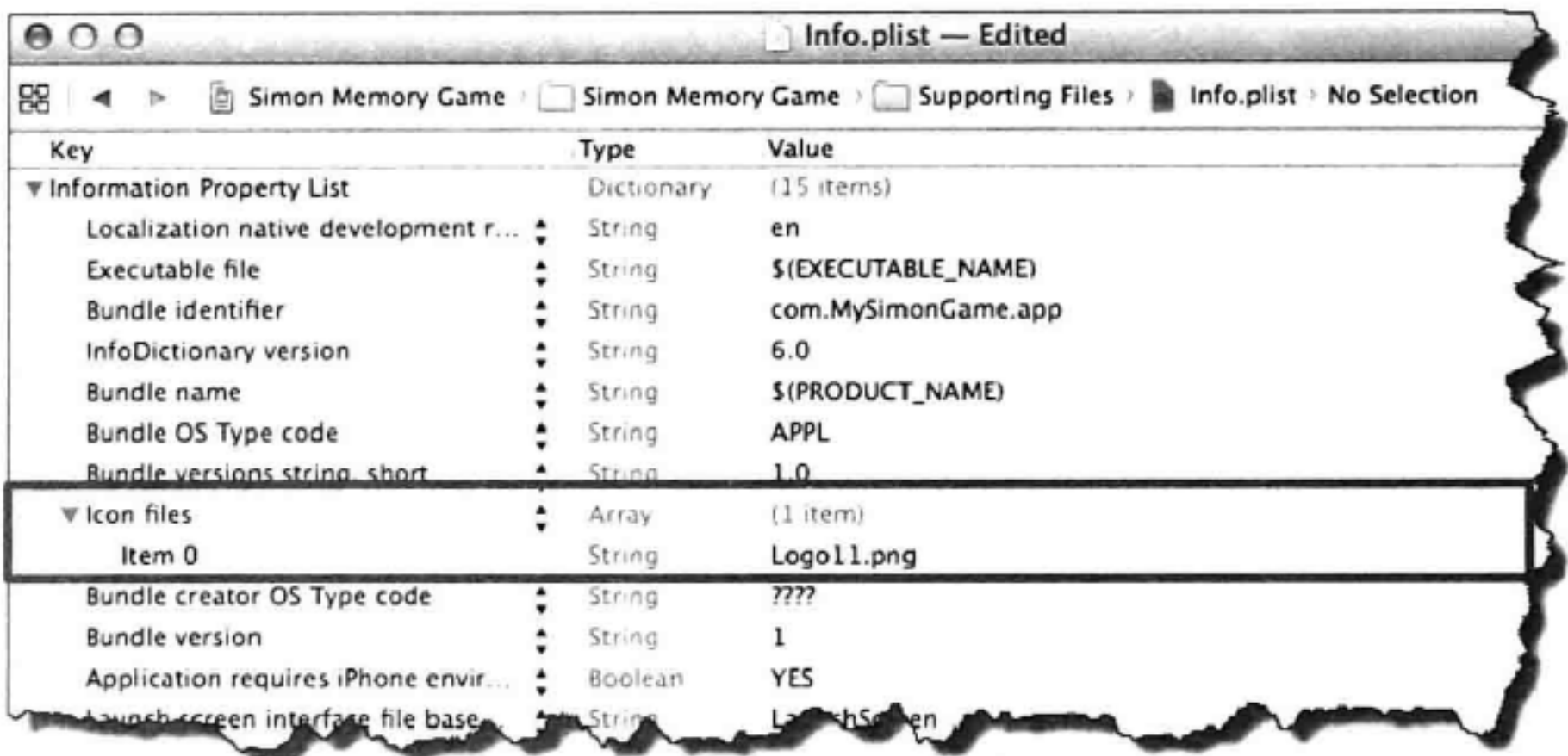


图 11.26 图标的设置

3. 设置Code Signing Identity

在目标窗口中选择 Build Settings 选项，在此选项中找到 Code Signing 的 Code Signing Identity，选择其中的 Release，将代码签名改为 iPhone Distribution:(申请的证书名称)，如图 11.27 所示。

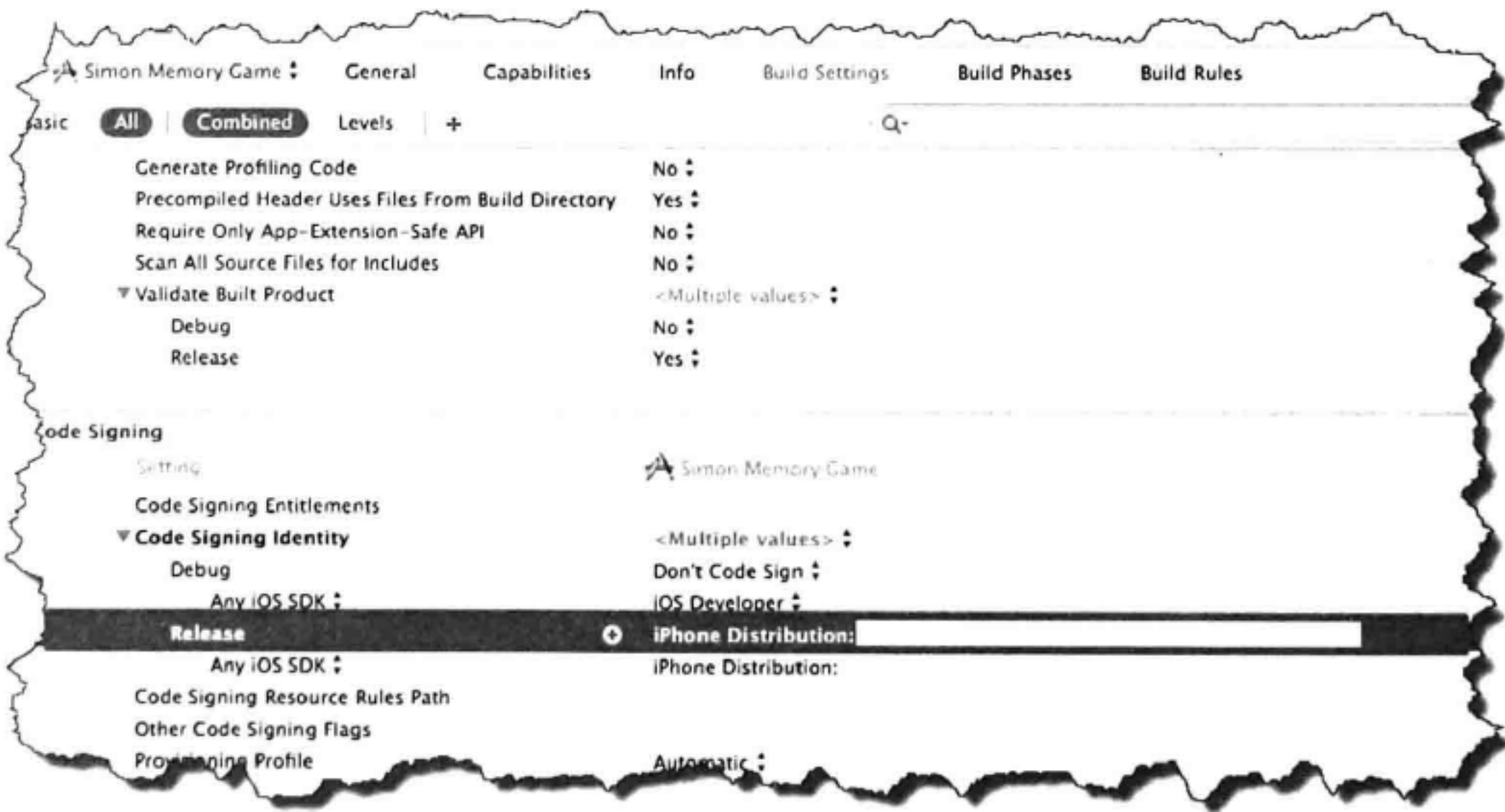


图 11.27 设置 Code Signing Identity

4. 发布编译

以下就发布编译的具体操作步骤。

(1) 选择工具栏上的 Product|Scheme|Edit Scheme...命令，打开编辑 Scheme 对话框，如图 11.28 所示。

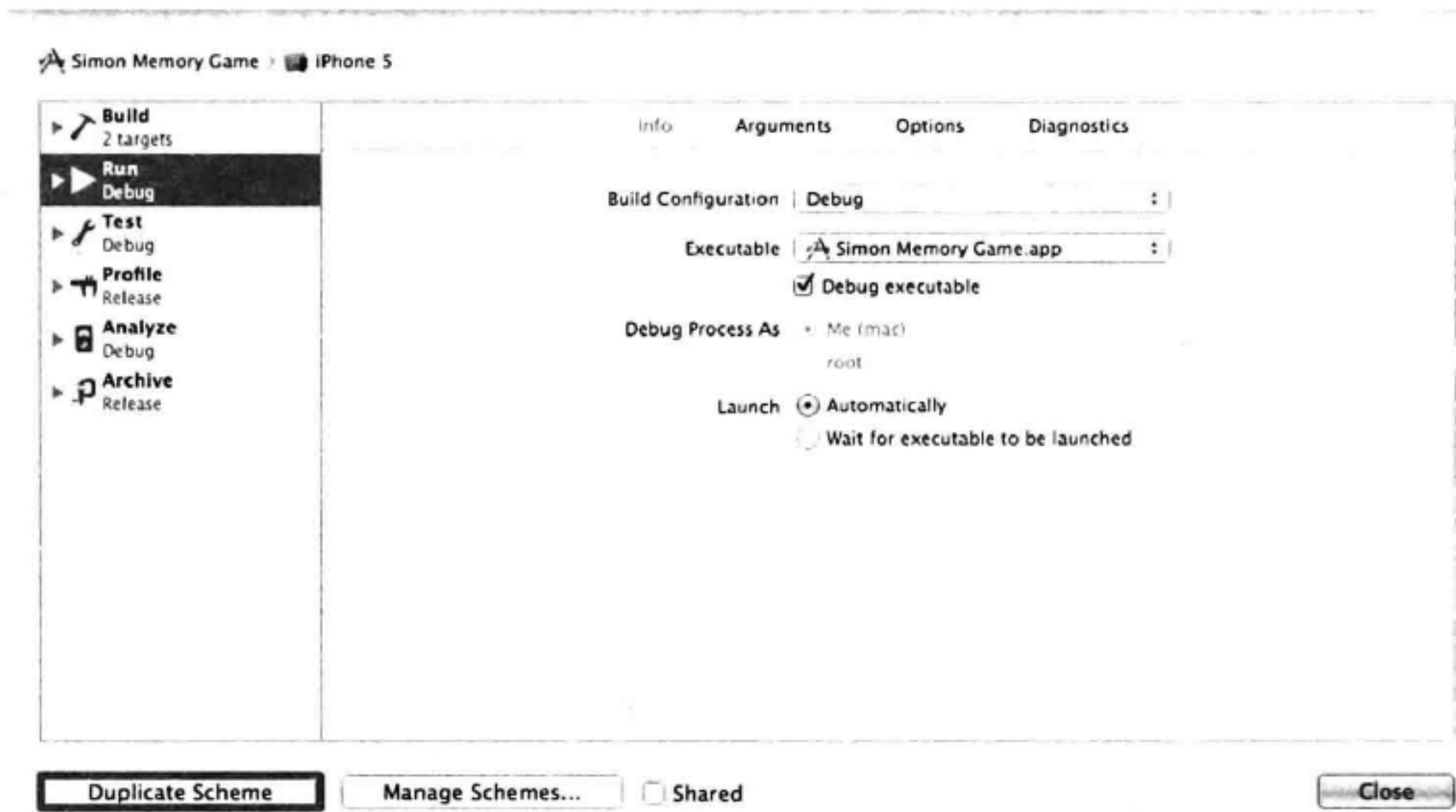


图 11.28 操作步骤 1

(2) 单击 Duplicate Scheme 按钮，复制一份新的 Scheme 为 Copy of Simon Memory Game。

(3) 在左上角的 Scheme 下拉列表中选择 Copy of Simon Memory Game 选项，选择左边列表中的 Run，在右边选择 Info 选项，在 Build Configuration 下拉列表中选择 Release 选项，如图 11.29 所示。

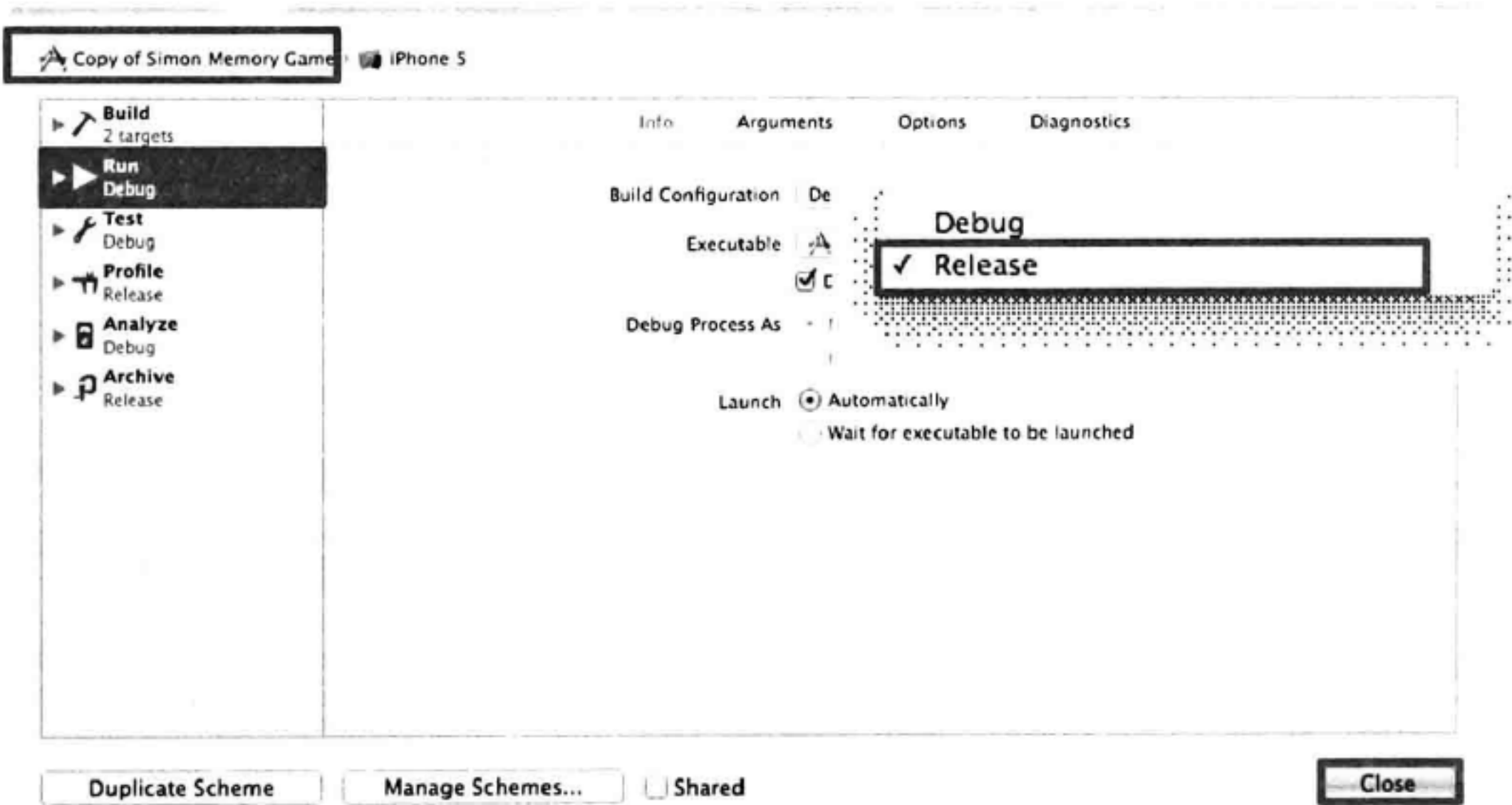


图 11.29 操作步骤 2

(4) 选择 Copy of Simon Memory Game 中的 iOS Device，如图 11.30 所示。

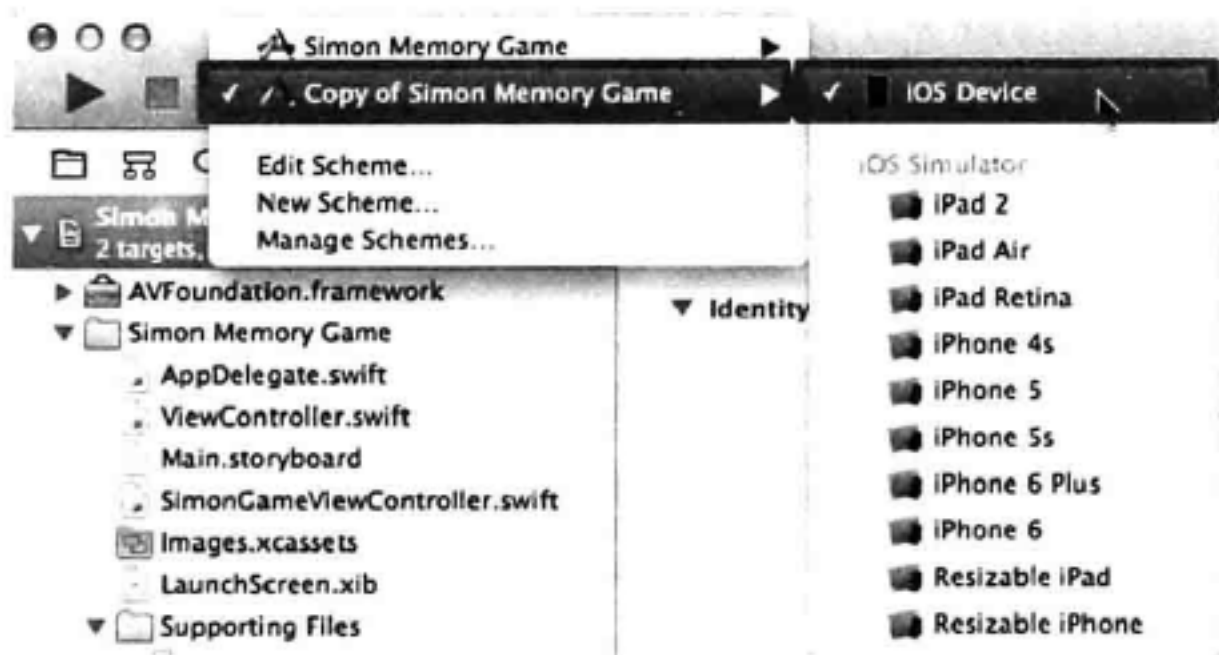


图 11.30 操作步骤 3

(5) 选择 Product|Build For|Runing 命令，就可以编译了。

注意：如果编译有错误或者发出警告，必须要解决，忽略警告往往会导致发布失败。在发布编译成功后，打开日志导航面板，会看到刚刚执行的 Copy of Simon Memory Game 已经成功了，如图 11.31 所示。



图 11.31 发布编译成功

5. 打包应用程序

在上传应用程序到 App Store 之前，我们需要将编译的二进制文件和资源文件打成压缩包，压缩的格式为 zip。以下是打包应用程序的具体步骤。

(1) 找到编译的二进制文件和资源文件，这个是很重要的。也是不太好找的。我们首先需要回到图 11.31 所示的编译日志中，在其中找到 Create universal binary Simon Memory Game...这个内容，然后将它展开，展开后的内容如下：

```
CreateUniversalBinary
/Users/mac/Library/Developer/Xcode/DerivedData/Simon_Memory_Game-ajincn
hpwveztwdvflshnmuduguh/Build/Products/Release-iphonios/Simon\ Memory\
Game.app/Simon\ Memory\ Game normal armv7\ arm64
```

```
cd "/Users/mac/Desktop/Simon Memory Game"
export
PATH="/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.pla
tform/Developer/usr/bin:/Applications/Xcode.app/Contents/Developer/usr/
bin:/usr/bin:/bin:/usr/sbin:/sbin"
/Applications/Xcode.app/Contents/Developer/Toolchains/Xcode Default.
xctoolchain/usr/bin/lipo -create /Users/mac/Library/ Developer/Xcode/
DerivedData/Simon_Memory_Game-ajincnhpwveztwdvflshnmuduguh/Build/Interm
ediate/Simon\ Memory\ Game.build/Release-iphoneos/Simon\ Memory\
Game.build/Objects-normal/armv7/Simon\ Memory\ Game
/Users/mac/Library/Developer/Xcode/DerivedData/Simon_Memory_Game-ajincn
hpwveztwdvflshnmuduguh/Build/Intermediates/Simon\ Memory\
Game.build/Release-iphoneos/Simon\ Memory\
Game.build/Objects-normal/arm64/Simon\ Memory\ Game -output
/Users/mac/Library/Developer/Xcode/DerivedData/Simon_Memory_Game-ajincn
hpwveztwdvflshnmuduguh/Build/Products/Release-iphoneos/Simon\ Memory\
Game.app/Simon\ Memory\ Game
```

-output 之后就是应用编译之后的位置了，其中/Users/mac/Library/ Developer/Xcode/DerivedData/Simon_Memory_Game-ajincnhpwveztwdvflshnmuduguh/Build/Products/R
elease-iphoneos/是编译之后生成的目录，如图 11.32 所示。



图 11.32 编译之后生成的目录

(2) 右击 Simon Memory Game.app 文件，在弹出的快捷菜单中选择“压缩"Simon Memory Game"”选项，如图 11.33 所示。这样就会在当前目录下生产 Simon Memory Game.zip 压缩文件，请将此文件保存好，我们会在后面使用到。



图 11.33 压缩 Simon Memory Game.app 包文件

11.4 提交应用程序到 App Store 上

提交应用程序到 App Store 上一般有两种方法：Application Loader 和 Archives，本节将对这两种方法一一进行讲解。

1. Application Loader

从 2010 年年中开始，开发者上传软件必须使用 Application Loader 这个 MAC 机上的应用程序。如果您安装了最新版的 Xcode 开发环境，对于在 4.2 及以上版本，Developer/Applications/Utilities 目录中已经有 Application Loader 程序，无需执行单独安装。对于 Xcode 4.3 及以后版本，在/Applications/XCode.app/Contents/Applications 目录中可以找到（右键 Xcode 选择 Open Developer Tool 也可以看到 Application Loader）。以下是使用 Application Loader 上传游戏应用程序的具体步骤。

（1）右键 Xcode 选择 Open Developer Tool 选项，会看到 Application Loader，如图 11.34 所示。




图 11.34 操作步骤①

（2）选择 Application Loader，会弹出“登录”对话框，如图 11.35 所示。



图 11.35 操作步骤②

 **注意：**当开发者第一次使用 Application Loader 程序，会看到“Application Loader 软件许可协议”对话框，如图 11.36 所示。单击此对话框中的“同意”按钮，就会进入“登录”对话框。

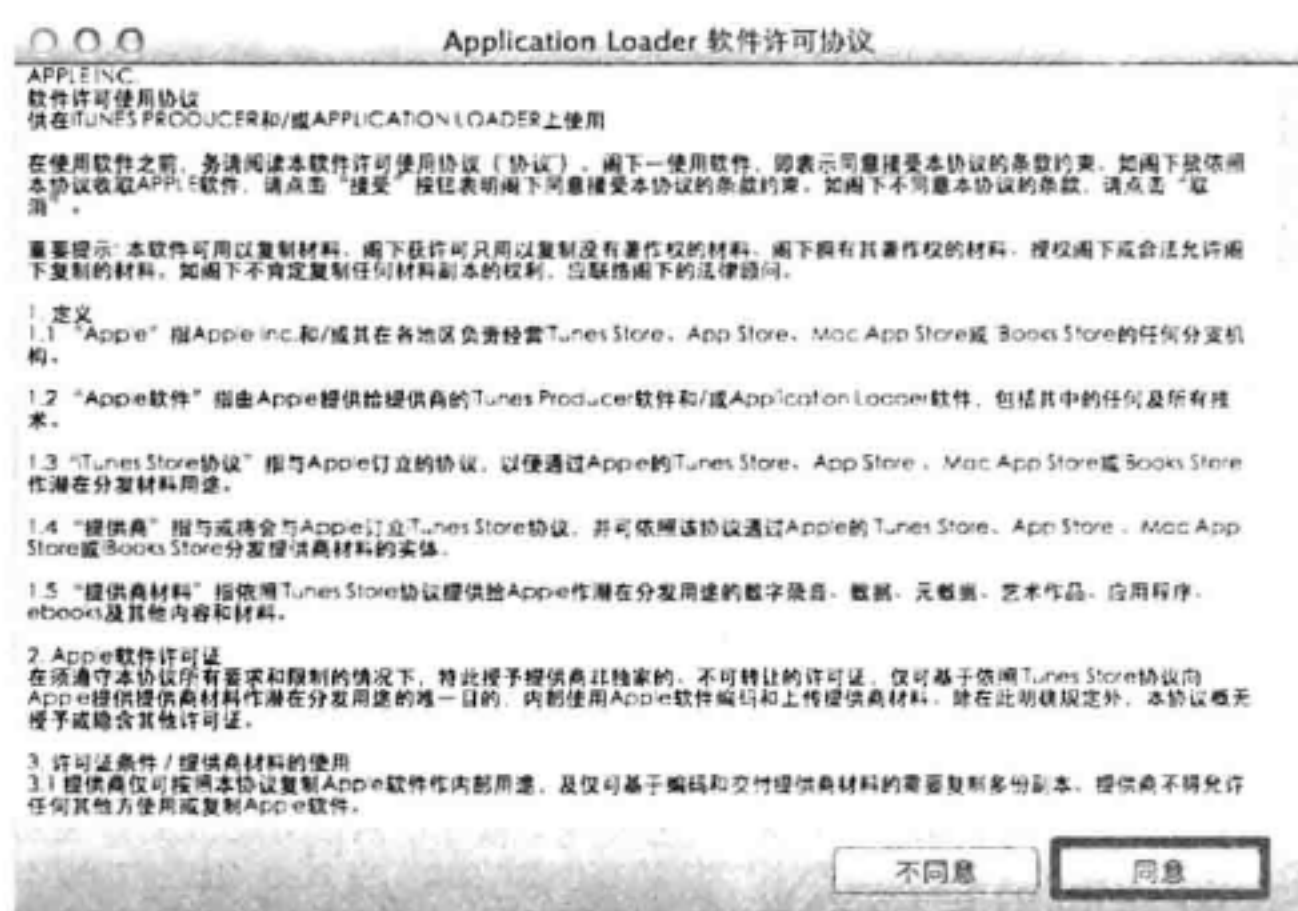


图 11.36 软件许可协议

(3) 输入苹果账号和密码后，单击“登录”按钮，会进入“模板选取器”对话框，如图 11.37 所示。

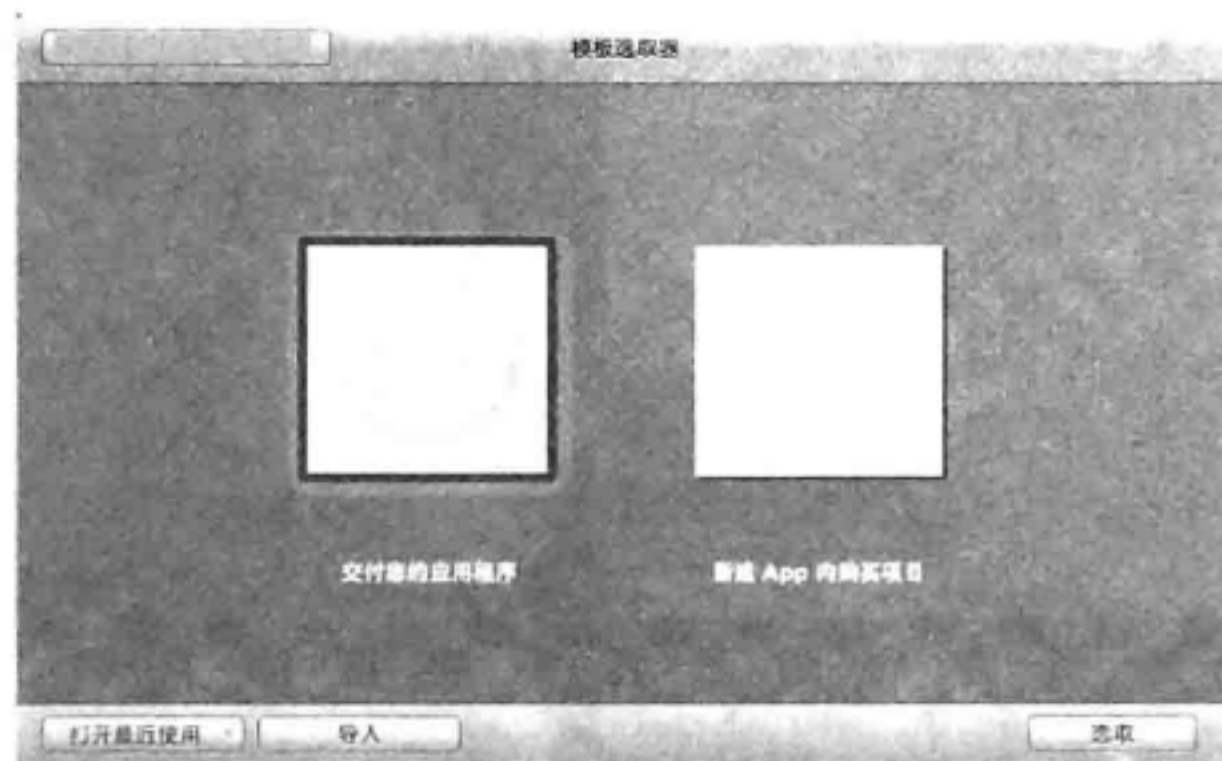


图 11.37 操作步骤 3

(4) 选择交付您的应用程序图标，会弹出选择文件对话框，如图 11.38 所示。



图 11.38 操作步骤 4

(5) 选择打包的 Simon Memory Game.zip 文件后，单击“打开”按钮，弹出有关应用程序信息的对话框，如图 11.39 所示。



图 11.39 操作步骤 5

(6) 单击“下一步”按钮，弹出“正在添加应用程序...”对话框，如图 11.40 所示。



图 11.40 操作步骤 6

(7) 当添加完成后，会出现如图 11.41 所示的对话框。



图 11.41 操作步骤 7

(8) 单击“下一步”按钮，弹出“谢谢您”对话框，如图 11.42 所示。



图 11.42 操作步骤 8

此时应用程序就提交到了 App Store 上，这时就是等待审核的状态了，审核通过后，应用程序就发布成功了。

2. Archives

Archives 是 Xcode 现在推行使用的上传方式，以下是使用 Archives 上传游戏应用程序的具体操作步骤。

(1) 选择 Product|Archive 命令，打开 Organizer-Archives 对话框，如图 11.43 所示。

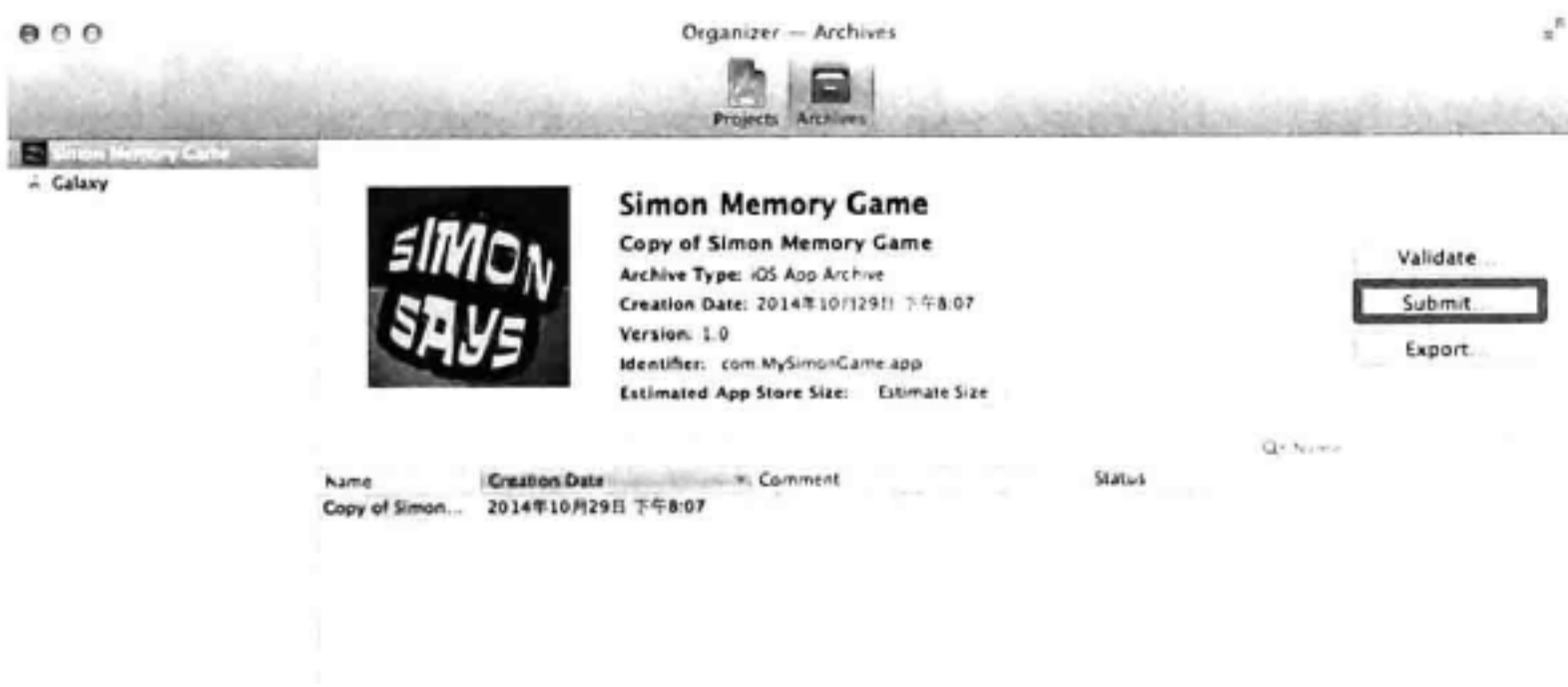


图 11.43 操作步骤 1

(2) 选择 Submit...按钮后，弹出获取开发团队的对话框，如图 11.44 所示。一段时间后，此对话框将变为选择开发团队对话框，如图 11.45 所示。

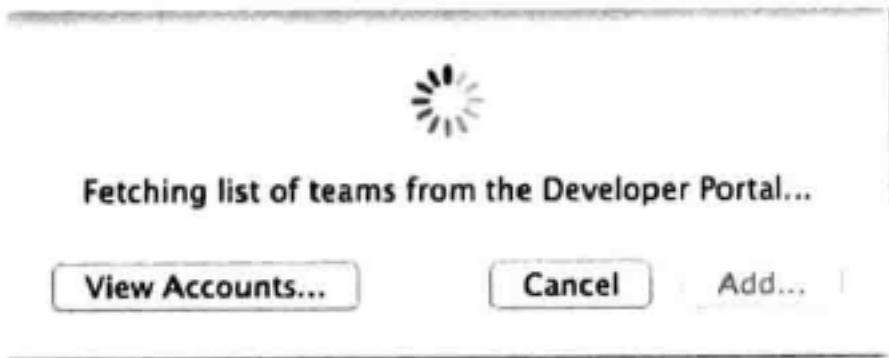


图 11.44 操作步骤 2

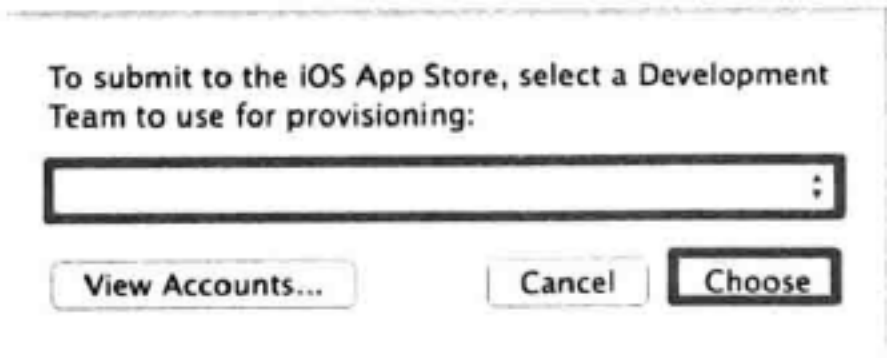


图 11.45 操作步骤 3

(3) 选择开发团队，并单击 Choose 按钮，弹出 Send Copy of Simon Memory Game to Apple:对话框，如图 11.46 所示。



图 11.46 操作步骤 4

(4) 选择 Submit 按钮后，弹出 Preparing archive for submission 对话框，一段时间后此对话框变为了 Submitting archive to the iOS App Store:对话框，如图 11.47 所示。



图 11.47 操作步骤 5

(5) 提交完成后，弹出 Archive submission process complete:对话框，如图 11.48 所示。



图 11.48 操作步骤 6

11.5 常见审核不通过的原因

App Store 的审核是很严格的。苹果官方提供了一份详细的审核指南，包括 22 大项，100 多小项的拒绝上线条款，并且条款还在不断地增加。此外，还包括一些模棱两可的条例，所以稍微有“闪失”，应用就有可能被拒绝。以下就是应用被拒绝的常见原因。

1. 功能问题

在开发应用前，开发者一定要对产品进行认真的测试。如果你的应用程序存在崩溃、错误、使用非公开 API、有意提供隐蔽或虚假功能，却又不能明显标示的问题，无疑是被审核小组拒绝的对象。

2. 界面问题


苹果审核指南规定，开发者的应用必须遵守《iOS 用户界面指导原则》中解释的所有条款和条件，如果违反这些规则，就会拒绝上线。

3. 商业问题

在发布应用时，首先不可以侵犯苹果公司的商标和版权。也就是在应用中不能出现苹果的图标，不能使用与苹果公司现有产品相似的名称为应用命名。

4. 不当内容

一些不合适、不合法的内容，苹果公司也不允许上架，如涉嫌诽谤、侮辱、狭隘内容或打击个人或团体的应用、展示人或动物被杀戮、致残、枪击、针刺或其他伤害的真实图片的应用、描述暴力或虐待儿童的应用、含有韦氏词典中定义的色情素材的应用等。

 **注意：**Steve Jobs 非常在意 App Store 的色情内容。他曾说：“如果你想要色情内容，那么就用 Android 手机吧”。

5. 其他

除以上这些内容被拒外，还有位置、推送通知、iAD 相关的、媒体内容、购买与流通、抓取和聚合、设备损害、暴力等存在的问题也会被拒。

 **说明：**详细内容请开发者参考 iOS APP 审核规则（<https://developer.apple.com/app-store/review/>）。